VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Paralelní přístupy v kompresi dat

Parallel approaches in data compression

2018 **Bc. Ondrej Papaj**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student:	**Bc. Ondrej Papaj**
Studijní program:	N2647 Informační a komunikační technologie
Studijní obor:	2612T025 Informatika a výpočetní technika
Téma:	Paralelní přístupy v kompresi dat Parallel Approaches in Data Compression
Jazyk vypracování:	čeština

Zásady pro vypracování:

Cílem práce je navrhnout paralelní architekturu pro kompresi masivních dat za pomocí více jádrových procesorů i více uzlového serveru. Hlavním bodem bude navržení částí, které je možné paralelizovat u dnes nejpoužívanějších kompresních algoritmů.

Práce musí obsahovat:
1. Aktuální stav v oblasti paralelní komprese dat.
2. Analýzu použité kompresní metody se zaměření na paralelizovatelné části.
3. Návrh architektury pro paralelní kompresi.
4. Otestování efektivity a funkčnosti algoritmu.

Seznam doporučené odborné literatury:

[1] Data Compression: The Complete Reference, David Salomon, 4. ed., Springer, 2007
[2] Parallel Programming with MPI, Peter S. Pacheco, Morgan Kaufmann, 1997, ISBN 978-155-8603-394

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Ing. Jan Platoš, Ph.D.**

Datum zadání: 01.09.2014
Datum odevzdání: 30.04.2018

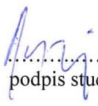
doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlášení studenta

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne: 29.06.2018



.....
podpis studenta

Poďakovanie

Rád by som poďakoval kolegovi Ing. Ladislavovi Ivančovi, ktorý mi pomohol pri úskaliach programovacieho jazyka C/C++ a hlavne doc. Ing. Janu Platošovi PhD. za odbornú pomoc, železnú trpezlivosť a konzultácie pri vytváraní tejto diplomovej práce.

Abstrakt

V posledných rokoch zaznamenávame nárast počítačového výkonu hlavne čo sa týka počtu jadier v rámci jedného procesora. Preto je v dnešnej dobe veľmi populárne využívať výkon všetkých jadier. Z týchto dôvodov sa táto práca zaoberá problematikou paralelizácie bezstratových kompresných algoritmov. V práci popíšeme existujúce riešenia a prístupy a následne sa pokúsime vytvoriť vlastnú implementáciu. Nakoniec budú v práci všetky spomenuté algoritmy porovnané.

Kľúčové slová

bezstratová kompresia, MPI, openMP, paralelizácia, bzip2, xz

Abstract

In recent years, we've seen an increase in computer performance, especially in terms of the number of cores per processor. Therefore, it is now very popular to use the performance of all cores. For these reasons, this work deals with the problem of parallelization of lossless compression algorithms. We will describe existing solutions and approaches in the work and then try to create our own implementation. Finally, all the above-mentioned algorithms will be compared.

Key words

lossless compression, MPI, openMP, parallelization, bzip2, xz

Obsah

1 Úvod ... 5
2 PBZIP2 ... 6
2.1 Bzip2 ... 6
2.2 Paralelizácia ... 7
2.2.1 Fronta FIFO ... 8
2.2.2 Štruktúra outBuff ... 9
2.3 Synchronizácia pbzip2 ... 9
2.3.1 Producent ... 10
2.3.2 Konzument ... 10
2.3.3 Zápis na disk ... 11
2.3.4 Dekompresia ... 11
3 MPIBZIP2 ... 12
3.1 MPI ... 12
3.1.1 Základné funkcie ... 12
3.1.2 Komunikácia medzi procesmi ... 13
3.2 Paralelizácia ... 13
3.3 Dekompresia ... 15
3.4 Debugovanie MPI ... 16
4 LBZIP2 ... 17
4.1 Paralelizácia ... 17
4.2 Dekompresia ... 18
5 PXZ ... 19
5.1 OpenMP ... 19
5.2 Paralelizácia ... 19
5.3 Dekompresia ... 21
6 PLZIP ... 22
6.1 Paralelizácia ... 22
6.2 Dekompresia ... 22
7 PIXZ ... 24
7.1 Paralelizácia ... 24
7.2 Dekompresia ... 24
8 PIGZ ... 25

8.1 GZIP ... 25
8.1.1 Deflate ... 25
8.2 Kompresia ... 25
8.3 Paralelizácia ... 26
8.3.1 Dekompresia ... 27
9 Paralelizácia bzip2 ... 28
9.1 Návrh a implementácia ... 28
9.2 Optimalizácie a vylepšenia ... 29
9.3 Ďalšie možnosti paralelizácie bzip2 ... 29
9.3.1 Paralelizácia BWT ... 29
9.3.2 Paralelizácia Huffmanovho kódovania ... 31
10 Paralelizácia XZ ... 32
10.1 XZ a LZMA SDK ... 32
10.1.1 7zip ... 32
10.2 Návrh a implementácia ... 32
10.3 Použitie ... 38
11 Testy a analýza ... 39
11.1 Analýza ... 39
11.2 Testy ... 39
11.2.1 Algoritmy bzip2 a deflate ... 40
11.2.2 Algoritmus lzma2 ... 41
12 Záver ... 42
Literatúra ... 43
Prílohy ... 45

Zoznam použitých skratiek a termínov

Skratka/Termín	Význam
ASCII	americký štandardný kód pre výmenu informácií
VMS	operačný systém pre výkonné serverové počítače
BSD-style	šírenie zdrojových kódov, ktoré vyžaduje len uvedenie autora
DDT	nástroj pre debugovanie paralelných aplikácií
GDB	GNU project debuger – nástroj na ladenie sekvenčných programov
HPC	High performance computing – prostredie pre vysoký výkon
IDE	Integrated development enviroment – vývojové prostredie
LZMA	Lempel Ziv Markov Chain
MPI	Message passing interface
OS X	Operačný systém pre počítače Macintosh
POSIX	Portable Operating System Interface

Zoznam obrázkov

1 Sekvenčné Fázy bzip2 algoritmu 6
2 Paralelizácia bzip2.....8
3 Paralelizácia bzip2 pomocou MPI.....15
4 Sekvenčný diagram lbzip2 pri kompresii.....18
5 Rozdelenie súboru pri kompresii.....20
6 Čítanie súboru pomocou openMP.....29
7 Rýchlosť kompresie v závislosti od počtu jadier pre bzip a deflate.....40
8 Kompresný pomer v závislosti počtu jadier pre bzip2 a deflate.....40
9 Rýchlosť kompresie v závislosti od počtu jadier pre lzma2.....41
10 Kompresný pomer v závislosti počtu jadier pre lzma2.....41

1 Úvod

Kompresia dát hrala významnú rolu v oblasti výpočtovej techniky od 70. rokov 20.storočia, kedy sa internet stal populárnym a bol vynájdený algoritmus Lempel-Ziv. Má však oveľa dlhšiu históriu. Jej začiatky siahajú do roku 1838 kedy bola objavená Morseova abeceda. Nasledovalo Shannon-Fanove kódovanie a hneď za ním Huffmanovo. Potom prišli na radu algoritmy LZ77 a LZ78, ktoré boli viac menej prelomové[17]. Na ich základoch stoja takmer všetky súčasné nástroje(výnimku tvoria bzip2 a PPM).

V dnešnej dobe zaznamenávame obrovský nárast digitálnych dát a spolu s dátami rastie aj výpočtový výkon a to aj na osobných počítačoch. Nie je problém kúpiť notebook, ktorého procesor má niekoľko fyzických jadier. Tento trend dokonca prenikol aj do mobilných telefónov. Preto sa veľmi striedmo začínajú prispôsobovať aj aplikácie. Nejedná sa však o žiadny masový nástup viacvláknových nástrojov, ale používa sa hlavne v oblastiach kde je potrebný vysoký výpočtový výkon ako napríklad predpovede počasia, nukleárne simulácie, geologická seizmická aktivita atď.

Jedným z takýchto odvetví je určite aj kompresia dát. Počas zisťovania aktuálneho stavu v tejto oblasti sme zistili, že paralelizácia bezstratovej kompresie sa pomaličky dostáva do popredia. Príkladom môže byť nástroj ZStandart, ktorý keď sa začala písať(2017) táto práca ešte neponúkal viacvláknové spracovanie, ale dnes už je jeho súčasťou[20].

V práci najprv detailne rozoberieme prístupy paralelizácie algoritmov ako bzip2, gzip, xz, ktoré sú najviac používané na unixových systémoch. Okrajovo sa pozrieme aj na algoritmus lzma2, ktorý beží pod systémom Windows. Práca si kladie za cieľ navrhnúť architektúru pre viacjadrový respektíve viacuzlový server. Preto v najprv navrhneme paralelizáciu BWT a Huffmanovho kódovania. Práca obsahuje aj implementáciu bzip2 paralelizovaného pomocou openMP. Ďalej sa v práci nachádza hybridný prístup openMP a MPI, pomocou ktorého je paralelizovaná knižnica xz. Nakoniec sú nad všetkými algoritmami vykonané testy kompresného pomeru a rýchlosti.

2 PBZIP2

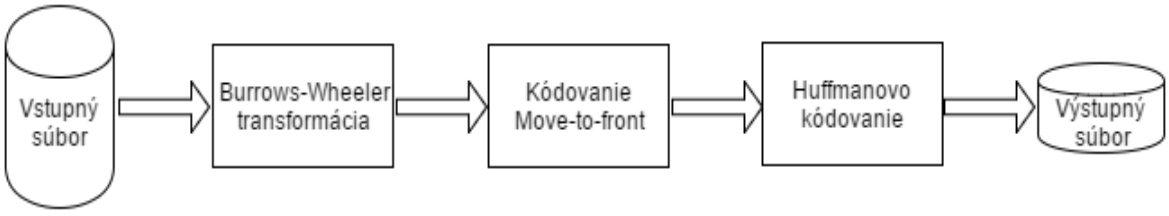
Paralell bzip2 používa pôvodný algoritmus "block-sorting" bzip2, ktorý je paralelizovaný pomocou knižnice pthread a dosahuje takmer lineárne zrýchlenie na SMP staniciach. Je ho možno používať na väčšine operačných systémov napr. Linux, Windows(za pomoci knižníc cygwin a MinGW), Solaris, OSX a distribuuje sa pod licenciou BSD-style. [1]

Jeho hlavným autorom je doktor Jeff Gilchirst. Hlavným dôvodom prečo začal doktor vyvíjať tento nástroj bol príchod dvojjadrových procesorov medzi bežných užívateľov. Vtedy bol bzip2 nový štandard na Linuxových systémoch a mohol byť oveľa lepší ako formát ZIP, ale bol časovo aj výpočtovo náročnejší. Tak začal pracovať na vývoji. Túto aktivitu si všimol Bulhar Yavor Nikolov, ktorý je aj hlavným prispievateľom. Yavor kontaktoval Jeffa s návrhmi a vylepšeniami a ponúkol sa, že bude písať aj zdrojový kód. Posledný release bol publikovaný 9. januára 2016 [1] čo znamená, že podpora tejto utility stále trvá. Podľa slov autora stále opravujú nahlásené chyby, ale k výrazným zmenám by už dôjsť nemalo.

2.1 Bzip2

Autorom je Julian Seward a prvá verzia vyšla v roku 1993. Je šírený ako open-source (BSD licencia) takže nie je zaťažený patentmi. Je teda možné ho bezplatne používať ale aj slobodne upravovať jeho verejne dostupný zdrojový kód. Bzip2 totiž ponúka programátorské aplikačné rozhranie, vďaka ktorému je možné čítať alebo zapisovať súbory vo formáte bz2. Túto možnosť využíva aj náš pbzip2.

Bzip2 používa kombináciu rôznych bezstratových kompresií. Vstupný súbor je rozdelený na rovnako veľké bloky dát, ktoré môžu byť spracované nezávisle. Každý blok je posunutý do rúry(pipeline) algoritmu. Najdôležitejšie fázy tejto rúry sú znázornené na obrázku č 1. Skomprimované bloky získané na konci rúry sú zoradené podľa originálneho poradia vo výstupnom súbore. Všetky transformácie sú vratné. Pre dekompresiu sú tieto fázy vykonávané v opačnom poradí [2].



Obrázok 1: *Sekvenčné Fázy bzip2 algoritmu*

Prvá fáza vykonáva Burrows-Wheelerovu transformáciu(BWT) na jednom bloku dát. BWT je algoritmus, ktorý preskupí znaky vstupného textu tak, že zhodné znaky budú pravdepodobne vedľa seba(tzn. že to nutne tak byť nemusí). Následne aplikuje Move-to-front filter , ktorý postupne zoberie znaky zo vstupu a na výstupe je napríklad kód z ASCII tabuľky. Ak je niekoľko za sebou nasledujúcich

znakov rovnakých tak sa prvý zakóduje a ostatné sa zakódujú na 0. Nakoniec sa použije Huffmanovo kódovanie kde znaky, ktoré sa vyskytujú najčastejšie majú najkratší kód.

2.2 **Paralelizácia**

Pbzip2 pracuje tak, že rozdelí vstupný súbor na bloky rovnakej veľkosti. Tým nastane mierne zrýchlenie aj pri jednom jadre, pretože sa číta v celom bloku naraz a nie postupne.

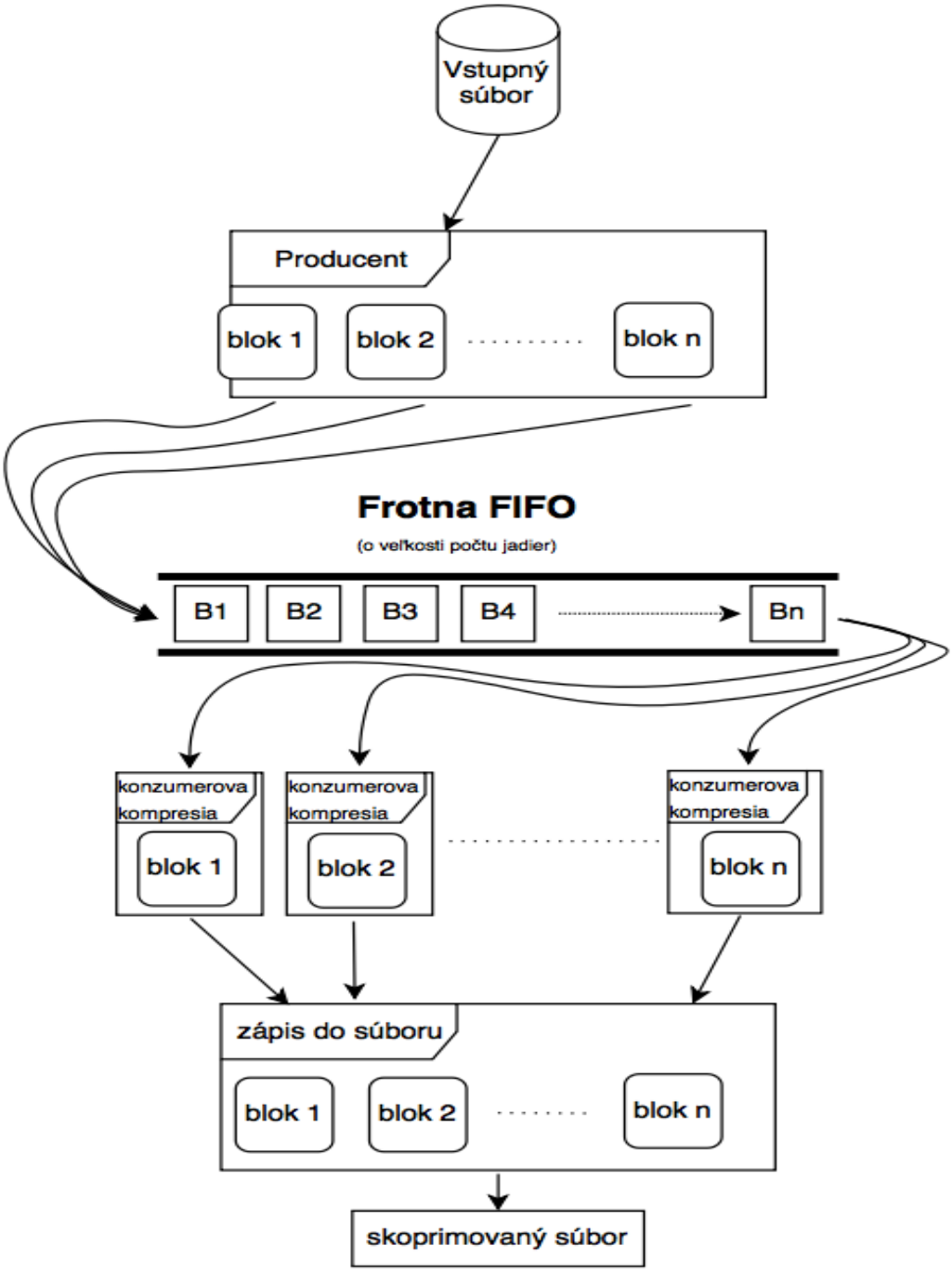
Jedným z prvých krokov je zistenie koľko procesorov respektíve jadier má stanica k dispozícii. Podľa počtu výpočtových jadier sa vytvorí fronta FIFO(first in - first out). Takáto veľkosť fronty nám dáva dobrú rovnováhu medzi rýchlosťou a veľkosťou pamäte, ktorá je potrebná na spustenie. Ak by bola fronta väčšia ako počet jadier, tak by nenastalo žiadne zrýchlenie len by sa zvýšili nároky na pamäť.

Fronta je v algoritme reprezentovaná ako producent-konzument model. V tomto prípade je producent jediné vlákno, ktoré generuje dáta a pridáva ich do fronty a konzument ich odoberá. Počet konzumentov pre tento algoritmus je rovný počtu jadier. Platí, že producent musí počkať, ak je fronta plná a konzumenti musia čakať pokiaľ je fronta prázdna.

Predtým než sa začne čítať vstupný súbor tak sa vytvoria konzumenti(funkcia *consumer*). Každý konzument beží vo vlastnom vlákne. Následne sa v hlavnom vlákne spustí producent(funkcia *producer*) a ten začne napĺňať frontu. Následne prídu na rad konzumenti. Tí postupne odoberajú a komprimujú jednotlivé bloky z fronty. Akonáhle je blok spracovaný tak sa jeho pamäť uvoľní a do globálneho vektoru *vector<outBuff>* sa uloží skomprimovaný blok. Pri uvoľnení miesta vo fronte producent znova pridá do fronty nový blok. Tento proces sa opakuje dovtedy kým nie je fronta prázdna. Všetky bloky sa komprimujú nezávisle na sebe.

V čase keď sú vytvorení všetci konzumenti sa vytvára nové vlákno, ktoré sa bude starať o zápis výsledného súboru na disk. Toto vlákno prechádza globálny vektor a zapisuje jednotlivé bloky v správnom poradí. Keď je zapísaný posledný blok program skončí.

Synchronizácia všetkých vlákien prebieha pomocou mutexov a podmienených premenných. Tie zaručujú, že k fronte bude mať prístup vždy len jedno vlákno. To znamená buď producent alebo jeden z konzumentov. Celá parelelizácia je znázornená na obrázku č 2.



Obrázok 2: *Paralelizácia bzip2*

2.2.1 Fronta FIFO

Jedným z kľúčových prvkov algoritmu je fronta FIFO. V algoritme je definovaná ako štruktúra. Je deklarovaná pomocou *typedef sturct* čo znamená, že tým vytvára vlastný dátový typ. Skladá sa z týchto atribútov:

`OutBuff *qData` – blok dát, ktorý sa bude komprimovať (dátový typ OutBuff je vysvetlený nižšie).

`long size` – je veľkosť fronty

`int empty` – indikuje či je fronta prázdna alebo nie

`pthread_mutex_t *mut` – toto je mutex, na základe ktorého môže k fronte pristupovať len jedno vlákno

`pthread_cond_t *notFull, *notEmpty` – podmienená premenná, na základe, ktorej sa čaká na splnenie podmienky pre uvoľnenie mutexu. V tomto prípade to je kontrola, či je fronta prázdna respektíve plná.

`pthread_t *consumers` – pole identifikátorov vlákien, ktoré ukazujú jednotlivých konzumentov, ktorý odoberajú bloky z fronty.

`outBuff *lastElement` – je to úplne posledný(najnovší) blok dát, ktorý bol do fronty pridaný

`Queue()` - konštruktor štruktúry, ktorý nastaví počet prvkov vo fronte a posledný blok na NULL

`void clear()` - vynuluje všetky dáta vo fronte

`Void add(outBuff element)` - pridá nový blok dát do fronty

`Void remove(outBuff &element)` - odstráni blok dát z fronty

2.2.2 Štruktúra outBuff

Táto štruktúra reprezentuje blok dát, ktorý sa vkladá do fronty a pri zápise na disk. Používa sa na ukladanie komprimovaných aj nekomprimovaných dát.

`Char* buff` – ukazuje na aktuálny blok ktorý sa bude komprimovať/dekomprimovať

`unsigned int buffSize` – veľkosť daného bloku

`int BlockNumber` – poradové číslo bloku, podľa ktorého sa zoradia skomprimované súbory.

`unsigned int inSize` – originálne veľkosť bloku, pred kompresiou

`int sequenceNumber` – ??

`bool isLastInSequence` – označuje či daný blok je posledný v sekvencii

`outBuf *next` – ukazuje na ďalší blok v poradí, ktorý sa bude komprimovať.

2.3 Synchronizácia pbzip2

Na vytváranie vlákien a ich synchronizáciu sa používa knižnica POSIX. Štandardy tejto knižnice sú určené pre programovanie pre prenositeľné unixové aplikácie. Aplikácie naprogramované vzhľadom na tieto štandardy budú prenositeľné medzi operačnými systémami, ktoré sú postavené na jadre unix ako napríklad Linux, FreeBSD, MacOS X. Operačný systém MS Windows neimplementuje toto rozhranie, ale existuje niekoľko riešení, ktoré umožňujú beh na tejto platforme.

2.3.1 **Producent**

Producent je reprezentovaný funkciou, ktorá sa volá z hlavného vlákna a má hlavičku:

```
int producer(int hInfile, int blockSize, queue *fifo)
```

Podľa návratovej hodnoty sa kontroluje, či prebehla funkcia správne.

`hInfile` – je vstupný súbor, ktorý sa bude komprimovať.

`blockSize` - veľkosť jednotlivých blokov,na ktoré sa rozdelí vstupný súbor.

`fifo` – fronta, do ktorej sa budbloky pridávať.

Pri vykonávaní tela funkcie sa spustí nekonečný cyklus WHILE1, kde sa zo vstupného súboru načíta prvý blok o zadanej veľkosti. Prebehne kontrola či načítane prebehlo v poriadku. Následne sa zamkne mutex, ktorý drží výhradný prístup k fronte a spustí sa cyklus WHILE2, ktorý na vstupnej podmienke kontroluje, či je fronta plná. Ak je fronta plná, tak pomocou podmienenej premennej zaradí aktuálne vlákno do fronty vlákien, ktoré čakajú na splnenie tejto podmienky – v tomto prípade sa kontroluje či je vo fronte voľné miesto. Cyklus testovania stavu podmienky *WHILE2* a volania *pthread_cond_wait()* je potrebný, pretože po ukončení čakania môže predbehnúť odblokované vlákno pri získaní výhradného prístupu k objektu iné vlákno a to môže stav podmienky zmeniť. Signalizáciu splnenia tejto podmienky má na starosti práve konzument. Po úspešnej kontrole fronty FIFO sa vytvára nový blok a to tak, že sa skonštruuje objekt(štruktúra) typu outBuff a ten sa následne vloží do fronty. Pomocou podmienenj premennej signalizujeme konzumentom, že fronta nie je prázdna. Inkrementuje sa počet blokov a odomkne sa mutex na fronte. Cyklus *WHILE1* sa vykonáva pokiaľ nie je prečítaný celý súbor.

2.3.2 **Konzument**

Konzumenta v algoritme reprezentuje funkcia, ktorá prijíma jeden argument a to ukazateľ na anonymný objekt, takže jej hlavička vypadá takto:

```
void *consumer (void *q)
```

Každé volanie tejto funkcie sa vykonáva vo vlastnom vlákne. Po zavolaní funkcie sa parameter pretypuje na frontu FIFO. Spúšťa sa prvý nekonečný cyklus FOR1, v ktorom sa zamkne mutex, ktorý drží výhradný prístup k fronte FIFO. Následne sa spustí ďalší nekonečný cyklus FOR2, v ktorom prebieha kontrola či nie je fornta prázdna a zároveň sa z nej odoberie blok dát pre kompresiu. Ak je fronta prázdna tak sa pomocou podmienenej premennej čaká na splnenie podmienky aby fronta FIFO nebola prázdna. Po splnení tejto podmienky, ktorej hodnotu nastavil producent, sa pokračuje nastavením

podmienky, na ktorú čaká producent a to, že fronta nie je plná. Po tejto kontrole sa uvoľní mutex a funkcia pristúpi ku kompresii dát zavolaním knižnice *libbzip2* konkrétne funkcie

```
int BZ2_bzBuffToBuffCompress(char* dest, unsigned int* destLen,
    char* source, unsigned int sourceLen, int blockSize100k,
                int verbosity,int workFactor)
```

kde

`char* dest` - adresa, na ktorej začína skomprimovaný výstup

`unsigned int* destLen` – dĺžka výstupu

`char* source` - adresa, kde začínajú vstupné dáta pre kompresiu

`unsigned int sourceLen` – dĺžka vstupných dát

`int blockSize100k` – veľkosť bloku pre Burrow-Wheelerovu transformáciu

`int verbosity` - nastavenie miery monitorovania/debugovania na výstupe

`int workFactor` – nastavenie ako sa zachová algoritmus pri rôznych typoch dát.

Po úspešnom vykonaní kompresie sa výsledok pridá do vektoru typu outbuff a nasleduje ďalšia iteráciu cyklu *WHILE1*.

2.3.3 **Zápis na disk**

Zápis na disk sa vykonáva v samostatnom vlákne(funkcia *fileWriter*). Spúšťa sa funkcia, ktorá ma jediný argument a to názov výstupného súboru. Všetko sa deje v jednom nekonečnom cykle *WHILE1* kde sa na disk ukladajú skomprimované bloky v správnom poradí. To znamená, že cyklus postupne prechádza globálny vektor a ak nejaký blok chýba tak počká na jeho vytvorenie.

2.3.4 **Dekompresia**

Dekompresia prebieha pomocou toho istého modelu ako kompresia. Najprv sa podľa počtu zadaných procesorov vytvorí daný počet producentov(funkcia *consumer_decompress*) následne sa v ďalšom vlákne zavolá funkcia *fileWriter* a nakoniec sa v hlavnom vlákne zavolá producent funkcia *producer_decompress.*

Producer_decompress pomocou funkcie *bz2StreamScanner.getNextStream()* prechádza všetky skomprimované bloky a pridáva ich do fronty(totožná ako pri kompresii).

Cosnumer_decompress postupne číta bloky z fronty a pomocou funkcie *BZ2_bzDecompress* komprimuje blok dát.

Funkcia *fileWriter* je totožná ako pri kompresii.

3 MPIBZIP2

MPIBZIP2 je paralelná implementácia bzip2 block-sorting kompresného nástroja, ktorý používa MPI a dosahuje značné zrýchlenie na klastrových strojoch. Výstup tohoto nástroja je kompatibilný s bzip2 v1.0.2 alebo novšej verzii. Tento nástroj môže pracovať na akomkoľvek systéme ktorý má C++ kompilátor podporujúci pthreads(napr. *gcc*)[3].

3.1 MPI

Message Passing Interface(rozhranie na výmenu správ) poskytuje prenosný a silný medzi procesorový komunikačný mechanizmus, ktorý zjednodušuje niektoré úskalia komunikácie medzi stovkami a až tisícami paralelne pracujúcich procesorov. Používa sa zväčša pri počítačových blokoch – zoskupeniach (computer clusters). MPI štandard bol vytvorený zhruba 60 ľuďmi zo 40 organizácií väčšinou z USA a Európy.

Štandardná špecifikácia MPI definuje knižnicu podprogramov, ktoré obsahujú komunikačné funkcie na prenos údajov medzi procesormi, funkcie vykonávajúce kolektívne operácie nad nejakou množinou procesorov, a mnohé iné funkcie, ktoré sa zaoberajú prenosom správ a dynamickým vytváraním nových procesov[4].

3.1.1 Základné funkcie

`MPI_Init(int *argc,chat **argv)` - inicializuje prostredie MPI. Žiadna MPI funkcia nemôže byť volaná pred touto funkciou.

`MPI_Finalize()` - ukončí prostredie MPI. Žiadna MPI funkcia nemôže byť volaná po tejto funkcii.

`MPI_Comm_rank( MPI_Comm comm, int *rank )` – vráti číslo procesu v rámci dané skupiny comm. Všetky procesy sú automaticky v skupine MPI_COMM_WORLD.

`MPI_Comm_size ( MPI_Comm comm, int *size )` - vráti počet procesov v skupine *comm*. Všetky procesy sú automaticky v skupine MPI_COMM_WORLD.

`MPI_Barrier ( MPI_Comm comm )` - synchronizačná bariéra všetkých procesov v skupine *comm*. To znamená že program pokračuje až keď všetky procesy zavolajú tento príkaz. Pre synchronizáciu všetkých procesov sa použije MPI_COMM_WORLD skupinu[5].

3.1.2 Komunikácia medzi procesmi

MPI používa niekoľko druhov komunikácie medzi procesmi - blokujúci/synchrónny, asynchrónny, s užívateľským bufferom, s istotou atď.

MPIBZIP2 používa len blokujúcu/synchrónnu komunikáciu. Tá funguje na základe posielania správ/dát medzi jednotlivými procesmi. Pre odoslanie správy sa používa:

`MPI_Send ( void *buf, int count, MPI_DataType datatype, dest, tag, MPI_Comm comm )` kde,

*buf - sú dáta správy

count - dĺžka správy (tzn. počet dt daného typu)

dataType maju typ MPI_Datatype (je možné použiť MPI_CHAR, MPI_BYTE, MPI_SHORT..atď

dest je adresát správy (číslo procesu v rámci skupiny comm) a skupinou comm.

tag každá správa môže mať priradený identifikátor pomocou tohto parametru.

Veľmi dôležite je, že aktuálny proces čaká kým sa správa nedoručí, dovtedy je blokovaný. Pre prijímanie správy sa používa

`MPI_Recv ( void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status )` - proces je blokovaný pokiaľ správa nedorazí. Správa je uložená v dĺžke count do bufferu buf (typ dát v správe je určený parametrom MPI_Datatype). Odosielateľ je určený parametrom source (číslo procesu v rámci skupiny comm). Typ správy je daný parametrom tag. Pre príjem správy od ľubovoľného odosielateľa, sa používa konštanta MPI_ANY_SOURCE. Pre príjem správy ľubovoľného typu, sa používa konštantu MPI_ANY_TAG[5].

3.2 Paralelizácia

Po inicializácii MPI a parsovaní vstupných argumentov sa program rozvetví na dve časti - Master a Slaves. Master ako hlavný proces vytvára nové vlákno *filewriter,* v ktorom prebieha zápis skomprimovaných dát. Master volá funkciu *producer* kde prebieha čítanie vstupného súboru. Vo vytvorených Slaves (*consumers*) prebieha komprimácia dát. Minimálny počet procesov, s ktorými je možné nástroj spustiť je 2. Fungovanie algoritmu je zachytené na obrázku č.3 sekvenčným diagramom.

Producer

Proces prebieha v cykle WHILE, ktorý skončí keď sú prečítané všetky dáta. V prvom prechode cyklu proces čaká pomocou funkcie MPI_Recv až pokiaľ nedostane od Slave prázdnu správu, že je/sú Slaves spustené. Funkcia MPI_Get_count zistí presný počet prijatých dát. V tomto prvom prechode sa zatiaľ neposielajú žiadne dáta. Ak MPI_Recv neprijal žiadne dáta tak sa preskočí zápis do výstupného

bufferu a pokračuje sa na funkciu MPI_Send, ktorá posiela vstupný blok dát do Slaves. Proces čaká, kým tieto dáta nejaký Slave neprijme. Ako tag sa posiela poradové číslo bloku, ktoré je dôležité pri zoraďovaní skomprimovaných blokov vo výstupom súbore.

Cyklus následne pokračuje v druhej iterácii v ktorej už prijme od Slave skomprimované dáta a uloží ich do poľa OutputBuff na pozíciu ktorá mu prišla v MPI_TAG. Cyklus skončí až je prečítaný celý vstupný buffer. Potom sa spustí ďalší cyklus WHILE, ktorý spracuje dáta od Salves, ktoré ešte bežali po prečítaní celého súboru.

FileWriter

V tomto vlákne sa komprimované bloky, ktoré sa nachádzajú v *Outputbuff* zapisujú na disk. Cyklus WHILE, ktorý končí keď sú zapísane všetky bloky ma na svojom začiatku podmienku, ktorá zaistí aby sa skomprimované bloky zapísali v správnom poradí:

```
if ((OutputBuffer.size() == 0) ||
      (OutputBuffer[currBlock].bufSize < 1)
      || (OutputBuffer[currBlock].buf == NULL))
```

kde

`OutputBuffer.size() == 0` kontroluje či už začal zápis skomprimovaných dát do *outputBuff*

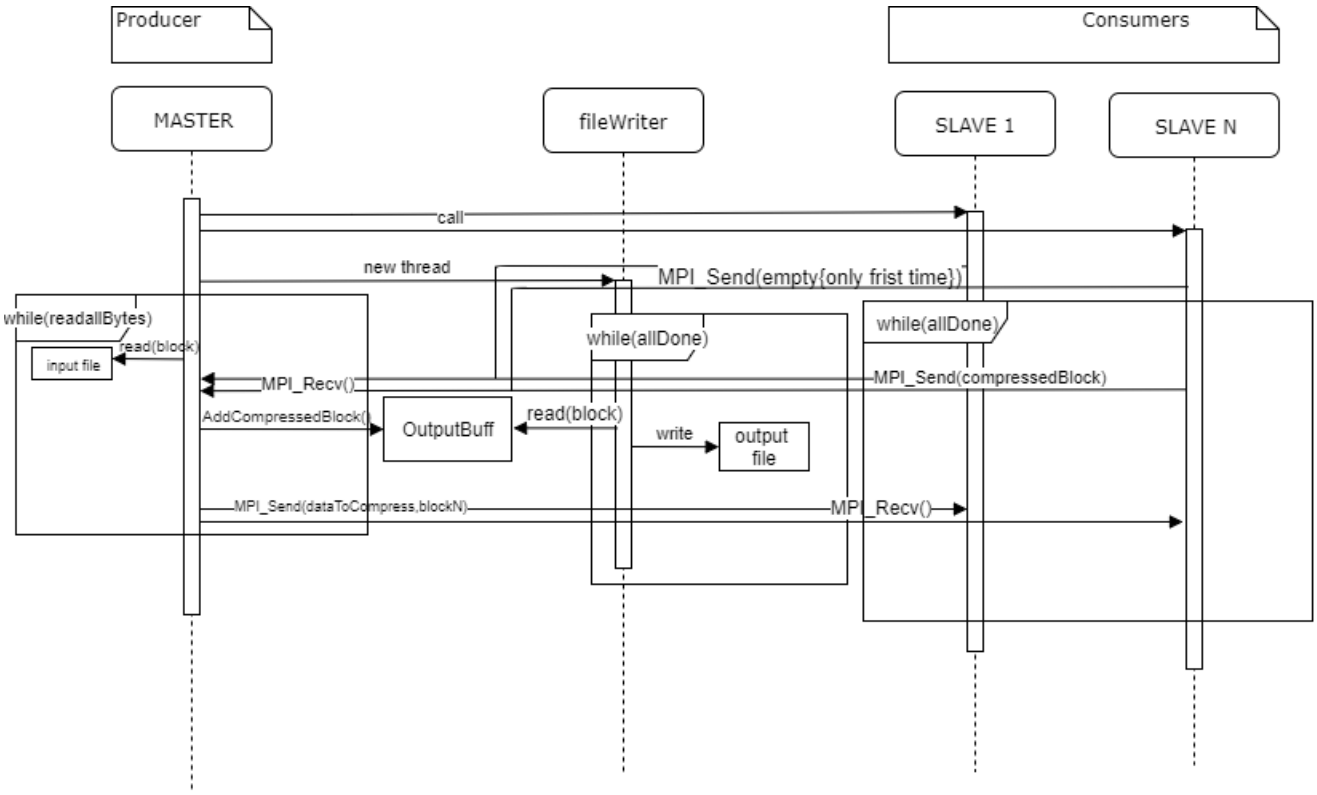
`OutputBuffer[currBlock].bufSize < 1` kontroluje či obsahuje nejaké dáta

`OutputBuffer[currBlock].buf == NULL` kontroluje či buffer obsahuje nejaké dáta.

Ak je táto podmienka splnená tak sa proces uspí a pomocou príkazu *continue* sa táto podmienka znovu overuje. V tomto cykle prebieha iterácia *currBlock* číslovaná od 0 takže cyklus čaká až sa naplní prvý blok *outputBuff*. Aby sa ušetrila pamäť tak po zápise sa hodnota tohto bloku nastaví na NULL.

Consumer

V prvom kroku, každý Slave pošle Mastrovi prázdny blok dát. Pre Mastra to znamená že Slave je pripravený. Ďalej nasleduje cyklus, ktorý končí, až keď sú spracované všetky bloky. V tomto cykle Slaves čakajú na dáta od Mastra pomocou funkcie *MPI_Recv().* Akonáhle prídu korektné dáta tak sa zavolá knižnica bzip2 konkrétne funkcia *BZ2_bzBuffToBuffCompress*, ktorá vráti procesu skomprimované dáta. Tieto dáta sú následne pomocou *MPI_Send()* poslané do Mastra a ten ich pomocou *filewriter* zapíše na disk.



Obrázok 3: *Paralelizácia bzip2 pomocou MPI*

3.3 Dekompresia

Prebieha na tom istom princípe ako kompresia až na malé rozdiely v producentovi a v konzumentoch:

Producent:

V tomto Master procese je čítanie rozdelené na dve fázy:

V prvej fáze sa prechádza celý súbor a hľadajú sa bzip2 hlavičky pre každý skomprimovaný blok. Je to z toho dôvodu, že pri komprimácii sa komprimoval každý blok samostatne a nezávisle na ostatných blokoch. Do vektoru sa postupne ukladá začiatok bloku a jeho veľkosť.

V druhej fáze sa tento vektor postupne prechádza a do konzumentov sa posielajú jednotlivé bloky.

Konzument:

Na dekompresiu sa volá funkcia *BZ2_bzBuffToBuffDecompress*. Táto funkcia môže vrátiť informáciu, že vstupná veľkosť bufferu, do ktorého sa ukladajú dekomprimované dáta je nedostačujúca.

V takom prípade proces prejde do cyklu, v ktorom sa v každej iterácii veľkosť bufferu zoštvornásobí až pokiaľ nie je jeho veľkosť dostačujúca. Následne sa skomprimované dáta pošlú do Master aby sa mohli zapísať na disk.

3.4 Debugovanie MPI

Pri hlbšej analýze algoritmu bolo potrebné program "oddebugovat". Veľa užívateľov používa *printf("")*, ktoré v niektorých prípadoch nie je dostačujúce. Preto existujú nástroje, ktoré umožňujú paralelné debugovanie ako napríklad DDT alebo TotalView. Obidva tieto silné nástroje sú platené preto bola zvolená tretia možnosť a to použiť sériový GDB debuger.

Každé vyspelejšie IDE ponúka možnosť pripojiť sa k lokálnemu procesu (*Attach to local process*). Aby bolo možné odchytiť proces vytvorený MPI je potrebné hneď za inicializáciu MPI dať nasledujúci fragment kódu:

```
MPI_Init(&argc,&argv);
int v = 0;
char hostname[256];
gethostname(hostname, sizeof(hostname));
printf("PID %d on %s ready for attach\n",getpid(),hostname);
fflush(stdout);
while(v==0)
   sleep(20);
```

Akonáhle sa program spustí pomocou mpirun tak každý proces najprv vypíše svoje pid na konzolu a následne ostane v nekonečnom cykle WHILE. Následne sa je možné pomocou *attach to local process* pripojiť k jednému z bežiacich procesov(ten s najnižším číslom býva v drvivej väčšine Master a ostatné sú Slaves). Potom už len stačí dať breakpoint na cyklus WHILE a proces sa zastaví. Pomocou *locals* sa zmení hodnota premennej *v* a podmienka cyklu nebude splnená a proces debugovania môže pokračovať. Tento návod je detailnejšie popísaný na stránkach open-mpi [6].

Na väčšine linuxových distribúcií je od určitej doby zakázaný *ptrace non-child* procesu *non-root-userom.* To znamená že debugovať proces môže len užívateľ s root právami a iba proces, ktorý daný podproces vytvoril. Ako dočasné riešenie stačí do terminálu zadať:

```
sysctl -w kernel.yama.ptrace_scope=0
```

4 LBZIP2

Lbzip2 je ďalší nástroj, ktorý používa paralelizáciu bzip2 pomocou POSIX vláknového modelu, ktorý mu umožňuje naplno využiť multiprocesorové (SMP) systémy(podobne ako PBZIP2). Tento nástroj je multiplatformný a je spustiteľný na širokej palete operačných systémov a viacerých hardvérových architektúrach. Je súčasťou najpopulárnejších GNU\Linux distribúcií[7].

Laszlo Ersek ho vytvoril hlavne preto, lebo vtedy neexistoval nástroj, ktorý by dokázal dekomprimovať súbor *.bz2, ktorý bol skomprimovaný jedným streamom, pomocou viacerých vlákien. Napríklad ak by sa komprimoval veľký súbor pomocou *standard bzip2* nie je možné ho dekomprimovať za použitia viacerých vlákien nástroja *pbzip2.* Autor si myslí, že do dnešného dňa neexistuje nástroj(okrem lbzip2), ktorý by toto dokázal.

Veľkou mierou do tohto projektu prispel aj Mikołaj Izdebski, ktorý naprogramoval low-level kompresiu bzip2, ktorá je značne rýchlejšia ako *standard bzip2.*

Podľa slov autora bude podpora projektu trvať dovtedy, kým bude tento nástroj užitočný a bude mať čas na jeho vývoj.

4.1 Paralelizácia

Nástroj je rozdelený do niekoľkých modulov. Prvý modul *main.c* má za úlohu rozparsovať argumenty a zistiť vstupné súbory. V ďalšom module *process.c* je pre každý vstupný súbor vytvorené vlákno *primary_thread*. Toto vlákno má na starosť tvorbu vlákien, ktoré budú pracovať systémom producent-konzument ale odlišným spôsobom ako tomu je pri pbzip2.

Primary_thread v prvom kroku inicializuje tri pracovné fronty pre vstupný súbor, ktoré sú implementované ako prioritné(priority queue). Ďalej inicializuje jednu cyklickú frontu(ring buffer) pre výstup. V ďalšom kroku sú vytvorené vlákna *sink_thread_proc* a *source_thread_proc.* Následne sa podľa vstupných parametrov vytvoria vlákna *worker_thread_proc,* kde 1x samotné *primary_thread* zavolá túto funkciu. Potom *primary_thread* pomocou funkcie *pthread_join()* čaká kým skončia všetky *worker_threads.* Nakoniec sa uvoľní pamäť zo všetkých front a do modulu *main.c* sa pošle sa signál SIGUSR2 o úspešnom konci.

Source_thread_proc v nekonečnom cykle postupne číta vo vstupnom súbore a naplňuje frontu *col_q.* V prípade, že je fronta plná tak toto vlákno čaká. Veľkosť fronty je nastavená na dvojnásobok počtu procesorov. Táto technika je známa pod pojmom double buffering. Keď kompresné vlákna(*worker_thread_proc*) komprimujú dáta, tak ďalší blok dát môže byť načítaný paralelne. Akonáhle kompresia bloku skončí ďalší blok už bude pripravený na kompresiu bez toho aby sa muselo čakať. Po prečítaní celého súboru vlákno skončí.

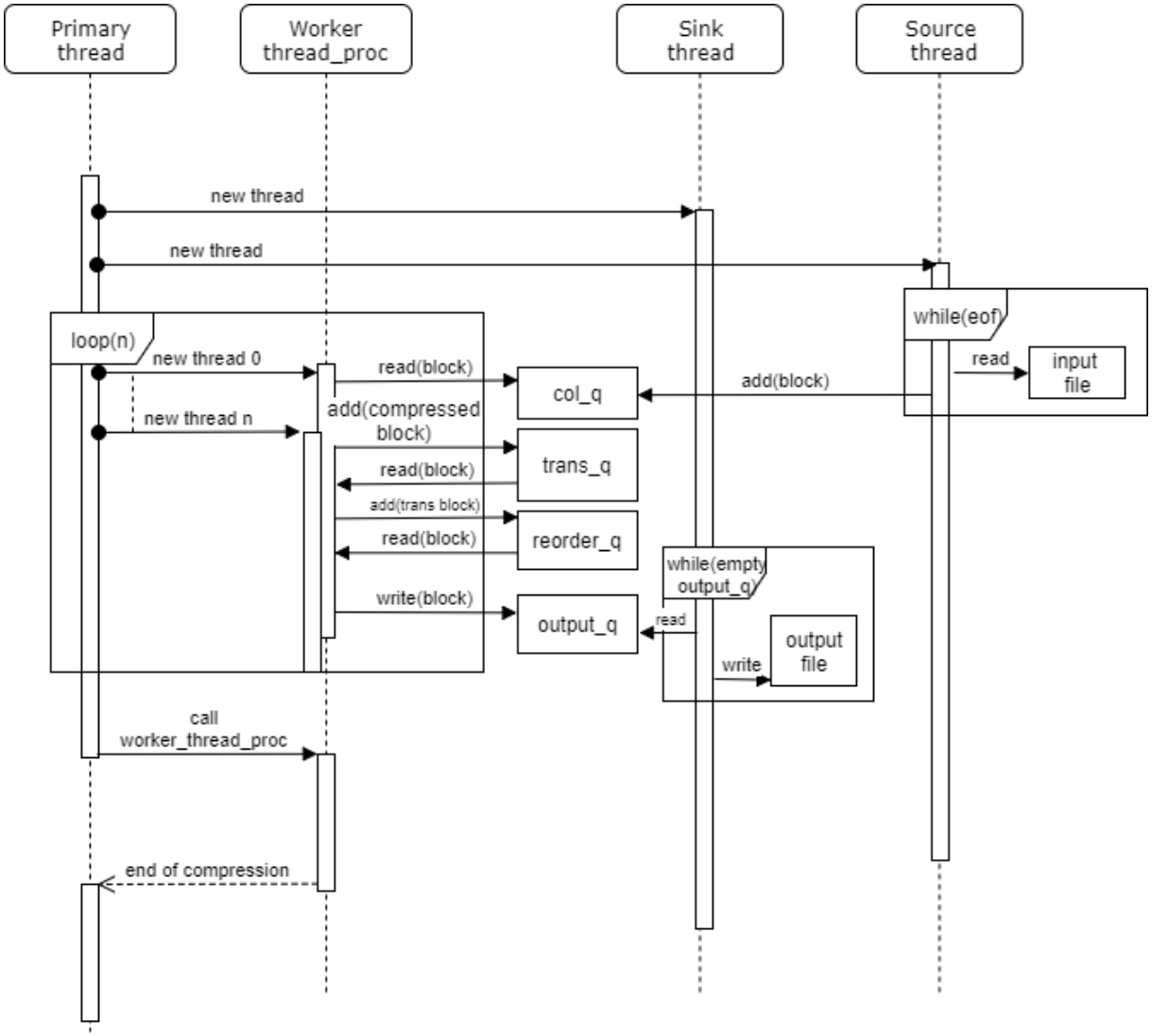
Worker_thread_proc v nekonečnom cykle sa spúšťajú tri úlohy(subtasks):

do_collect – v tejto funkcii sa číta z *col_q* blok dát, ktorý sa následne skomprimuje a výsledné informácie o kompresii uloží do fronty *trans_q*

do_transmit – načíta informáciey z *trans_q,* následne ich skompletuje do bloku a ten uloží do *reorder_q*

do_reorder – zoradene bloky uloží do *output_q*

Sink_thread_proc v nekonečnom cykle sa odoberá blok z *output_q* a zapisuje do výsledného skomprimovaného súboru.



Obrázok 4: *Sekvenčný diagram lbzip2 pri kompresii*

4.2 Dekompresia

Dekompresia prebieha rovnakým spôsobom ako kompresia.

5 PXZ

Paralel xz je nástroj na kompresiu, ktorý používa súčasný algoritmus LZMA na rôznych častiach vstupného súboru na viacerých jadrách alebo procesoroch. Jeho hlavným cieľom je využiť všetky zdroje na urýchlenie kompresného času s minimálnym možným vplyvom na kompresný pomer. Na paralelizáciu bola použitá technológia OpenMP. Autorom je Čech Jindřich Nový, ktorý tento nástroj vytvoril pretože vtedy neexistoval žiadny paralelný kompresný nástroj na báze LZMA a bz2 založený na BWT nedosahuje kompresného pomeru xz[8].

5.1 OpenMP

OpenMP je sústava direktív pre prekladač a knižnicových procedúr pre paralelné programovanie. Jedná sa o štandard pre programovanie počítačov so zdieľanou pamäťou. Uľahčuje vytváranie viacvláknových programov v programovacích jazykoch ako Fortran, C, C++. Hlavné vlákno *master thread* vytvára podľa potreby skupinu podvlákien. Paralelizácia programu sa potom uskutočňuje postupne s ohľadom na výkon aplikácie, tj. sekvenčný program je postupne paralelizovaný.[18]

Pre paralelizáciu programu sa používajú tzv. Direktívy:

```
#pragma omp parallel
{
  ...  // každé vlákno vykonáva príkazy tohto bloku
}
```

Nástroj pxz používa jedinú direktívu a to :

```
#pragma omp parallel for private(p) num_threads(threads)
```

`paralel for` – každé vlákno vykoná určitú časť iterácií.

`private(p)` - každé vlákno bude mať svoju kópiu premennej *p*

`num_threads(threads)` - nastaví koľko vlákien sa má vytvoriť[9].

5.2 Paralelizácia

Celá kompresia začína v cykle WHILE, ktorý končí vo chvíli, keď je prečítaný celý súbor.

Na začiatku cyklu sa vytvorí pole unikátnych dočasných súborov. Ich počet je určený počtom vlákien.

Ďalej sa do pamäte načíta prvá časť súboru, ktorej veľkosť závisí od veľkosti slovníka a od veľkosti _SC_PAGE_SIZE(veľkosť bloku v pamäti v bajtoch, ktorú alokuje funkcia mmap, jej veľkosť závisí od architektúry a typu stroja). Potom nasleduje samotná paralelizácia kde sa pomocou

#pragma omp parallel for

sparalelizuje cyklus FOR1 kde každé vlákno bude mať na starosť aspoň jednu iteráciu a cyklus skončí vtedy, keď sa spracuje aktuálna časť súboru, ktorá je práve uložená v pamäti. V každej iterácii cyklu FOR1 sa inicializuje *lzma_stream_encoder* a na základe poradia iterácie(iteruje sa pripočítavaním jednej) sa každému vláknu priradí aktuálna časť neskomprimovaného bloku. Ďalej sa pre každú iteráciu vytvorí cyklus FOR2, v ktorom aktuálne komprimovaný blok znova rozdelí na menšie časti. Veľkosť týchto častí je pevne nastavená na 65536(veľkosť posledného bloku môže byť menšia).Ako algoritmus rozdeľuje blok je možné vidieť na obrázku č.5. V tomto cykle FOR2 sa nastaví niektoré hodnoty *lzma streamu*:

`stream.netxt_in` – ukazateľ na ďalší vstupný bajt

`stream.avail_in` – počet vstupných bajtov

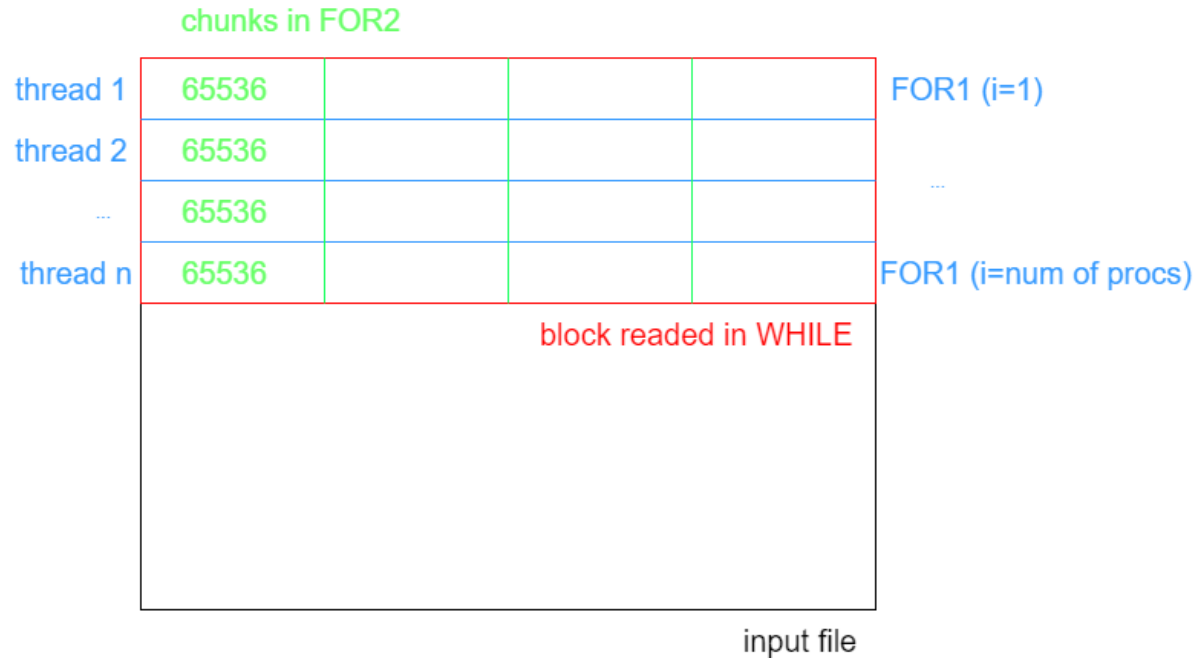
`stream.next_out` – ukazateľ na ďalší výstupný bajt

`stream.avail_out` – počet výstupných bajtov

Tento stream následne vstupuje do kompresie, ktorá prebieha pomocou funkcie:

```
Lzma_code(&stream, lzma_action)
```

kde *lzma_action* je akcia, ktorú ma knižnica *liblzma* so streamom vykonať. Výstup sa potom zapíše do predpripraveného dočasného súboru. Tu paralelizácia končí. Nakoniec sa v ďalšom cykle FOR čítajú všetky dočasné súbory a prebieha zápis do výstupného skomprimovaného súboru.



Obrázok 5: *Rozdelenie súboru pri kompresii*

5.3 Dekompresia

Algoritmus nepodporuje viacvláknovú dekompresiu.

6 PLZIP

Je masovo paralelný (viacvláknový) bezstratový algoritmus založený na kompresnej knižnici lzlib. Je spustiteľný na stovkách procesorov pre súbory o veľkosti niekoľko gigabajtov. Je oveľa rýchlejší ako lzip ale za cenu mierne zníženého kompresného pomeru(0.4 až 2% pri väčších súboroch). Výsledkom je súbor formátu lzip, ktorý je plne kompatibilný s nástrojom lzip 1.4 a jeho novšími verziami. Jeho autorom je Antonio Diaz Diaz, ktorý tento nástroj vytvoril aby slúžil ľudom zodpovedných za veľké množstvo dát najmä veľkých súborov a databáz. Pri porovnaní s PIGZ komprimuje efektívnejšie a dekomprimuje oveľa rýchlejšie[10].

6.1 Paralelizácia

Opäť je použitá knižnica POSIX pthread a model producent a konzument. Producentom je funkcia *csplitter* a konzumentom je *cworker.*

Trieda *Packet_curier*(jedná sa jediný objektový prístup vo všetkých analyzovaných algoritmoch) sa stará o prenos dát medzi producentom a konzumentom. Táto trieda obsahuje niektoré dôležité atribúty:

Packet - štruktúra, ktorá obsahuje dáta, ich veľkosť a poradové číslo

Packet_queue – fronta, kde sú uložené jednotlivé nespracované pakety

circular_buffer – vektor, v ktorom sú uložene skomprimované dáta

Slot_tally – trieda, ktorá sa stará o voľné sloty vo fronte

Obsahuje aj logiku vlákien tzn. mutexy, ktoré sa starajú o to aby k fronte pristupovalo vždy len jedno vlákno a podmienkové premenné, ktoré slúžia na predávanie signálov medzi jednotlivými vláknami. Ďalej obsahuje tieto metódy:

receive_packet získa slot z fronty a vloží tam blok neskomprimovaných dát.

distribute_packet - získava paket na komprimáciu

Collect_packet - vkladá skomprimovaný blok dát do vektoru

Deliver_packets - vracia zapisovacej funkcii zoradený vektor skomprimovaných blokov.

V každom vlákne *cworker* sa vyberie paket z fronty následne sa komprimuje a potom sa vloží do vektoru. Zapisovacia funkcia *muxer* sa v nekonečnom cykle WHILE volá funkcia *deliver_packet,* ktorej výsledok sa postupne zapisuje na disk.

6.2 Dekompresia

Pri dekompresii sa najprv prečíta celý súbor a vyhľadajú sa jednotlivé indexy, ktoré vznikli pri kompresii. Od počtu týchto indexov závisí aj počet vlákien, ktoré budú dekompresiu vykonávať. Ak je počet indexov menší ako počet zadaných vlákien tak počet vlákien bude rovný počtu indexov. Následne sa vytvoria dekomprimačné vlákna, ktoré sú reprezentované funkciou *dwork*. Ako argumenty tejto

funkcie je názov vstupného a výstupného súboru, id vlákna(poradové číslo vlákna v rámci cyklu FOR), v ktorom bola funkcia zavolaná, počet všetkých vlákien a jednotlivé indexy skomprimovaných blokov.

Vo funkcii *dwork* je cyklus for:

```
for(long i = worker_id;i < file_index.members();i += num_workers ),
```

ktorý na základe vlákna, v ktorom beží(worker_id) prechádza vo vstupnom súbore daný index. Aby k indexu pristúpilo len jedno vlákno tak je v cykle inkrementácia `i += num_workers`, ktorá spôsobí to že vlákno prekročí na index, ku ktorému sa iné vlákna nedostanú. Z tohto dôvodu nemôže byť počet vlákien väčší ako počet indexov. Ďalej sa v iterácii tohto cyklu postupne komprimuje daný blok a následne sa aj zapisuje do výstupu, pretože vďaka indexu vieme jeho pozíciu aj veľkosť pred kompresiou.

7 PIXZ

Je paralelná indexovacia verzia kompresného nástroja xz(vyslovuje sa *pixie*). Existujúce xz nástroje poskytujú skvelú kompresiu ale produkujú jeden veľký blok skomprimovaných dát. Pixz vytvára kolekciu menších blokov, ktoré umožňujú náhodný prístup k pôvodným dátam(indexácia). To znamená, že nepotrebujeme dekomprimovať celý súbor, ale len tú časť, ktorú potrebujeme. Na rozdiel od pxz poskytuje paralelnú dekompresiu[11]. Autora sa nepodarilo kontaktovať, takže jeho motivácia je neznáma.

7.1 Paralelizácia

Na paralelizáciu je opäť použitá knižnica POSIX threads a model producent-konzument. Najprv sa vytvoria 3 fronty *gPipelineStartQ*, *gPipelineSplitQ, gPipelineMergeQ*. Každý blok fronty obsahuje poradové číslo, ukazateľ na další blok a konkrétne dáta. Potom sa vytvoria konzumenti*(encode_thread),* ktorých počet závisí od vstupu užívateľa a len jeden producent *(read_thread)*.

Fronta *gPipelineMergeQ* slúži ako prostredník medzi producentom a konzumentmi. *Read_thread* postupne ukladá bloky do fornty a *encode_thread* ich v nekonečnom cykle vyberá z tejto fronty a komprimuje a výsledok ukladá do gPipelineMergeQ.

Hlavné vlákno sa stará o zápis skomprimovaných dát na disk. V nekonečnom cykle WHILE sa volá funkcia *pipeline_merged(),* ktorá čaká až sa vo fronte *gPipelineMergeQ* objaví blok s poradovým číslom 0 následne na blok číslo 1 až blok číslo n. Funkcia tento blok vráti a ten je zapísaný do výsledného súboru.

7.2 Dekompresia

Prebieha podľa tých istých postupov ako kompresia.

8 PIGZ

Tento algoritmus je paralelná implementácia nástroja gzip (GNU Zip), ktorá využíva výhody viacjadrových a viacprocesorových počítačov. Momentálne je dostupný len na unixových systémoch, ale je naprogramovaný tak aby mohol byť v budúcnosti *portovateľný* na Windows, VMS atď. Bol vytvorený doktorom Markom Adlerom, ktorý je spoluautorom pôvodného nástroja gzip a knižnice zlib. [12] Medzi jeho ďalšie zaujímavé projekty patrí spolupráca na vývoji obrázkového formátu PNG a vedúca pozícia na projekte vesmírneho vozítka *Mars Exploration Rover*[13]. Podľa slov autora bol hlavnou motiváciou pre vytvorenie pigz dopyt užívateľov využiť výhody ich viacjadrových procesorov. Autor bude vyvíjať tento produkt ešte niekoľko rokov a keď bude dostatočne zrelý tak ho predá ďalším programátorom.

8.1 GZIP

Gzip vznikol ako náhrada za už zastaralý unixový *compress,* pričom vo väčšine prípadoch dosahuje lepších kompresných pomerov a je zbavený všetkých problémov s patentami. Po čase si ich vzal pod seba GNU projekt a tak sa stal aj súčasťou linuxových distribúcií. Jeho autormi sú Jean-loup Gaillz a Mark Adler.

GZIP je založený na algoritme DEFLATE, ktorý je kombináciou LZ77 a Huffmanovho kódovania. DEFLATE bol určený k nahradeniu LZW a ďalších patentom zaťažených algoritmov.[14]

8.1.1 Deflate

Komprimované dáta sa skladajú zo sérií blokov, kde veľkosť blokov je ľubovoľná. Každý blok je komprimovaný kombináciou algoritmu LZ77 a Huffmanovho kódovania. Huffmanovým kódovaním sa vytvárajú Huffmanové stromy, ktoré sú pre každý blok nezávislé od predchádzajúcich a nasledujúcich blokov. Algoritmus LZ77 môže použiť odkaz na duplicitný reťazec, vyskytujúci sa v niektorom predchádzajúcom bloku. Každý blok sa skladá z dvoch častí. Jednu časť tvorí dvojica Huffmanových stromov, ktorá popisuje zastúpenie komprimovanej časti dát. Druhú časť tvoria komprimované dáta. To znamená, že Huffmanove stromy sa ukladajú spolu s komprimovanými dátami. Komprimované dáta sa skladajú z odkazov na duplicitný reťazec, vyskytujúci sa v niektorom predchádzajúcom bloku, a reťazcov, ktoré neboli zistené ako duplicitné v predchádzajúcich vstupných bytoch. Odkaz je určený dĺžkou reťazca a spätnej vzdialenosti jeho umiestnenia. Limit spätnej vzdialenosti je 32 kB. Každý typ hodnoty v komprimovaných dátach je reprezentovaný pomocou Huffmanovho kódu, používajúci jeden kódovací strom pre reťazce a ich dĺžku a druhý samostatný kódovací strom na vzdialenosť.[15]

8.2 Kompresia

Pigz vyžaduje pre svoj beh knižnicu zlib minimálne vo verzii 1.2.1. Vstup je rozdelený na rovnaké 128KB časti. Každá táto časť je skomprimovaná paralelne. Kompresia produkuje čiastočné *deflate streamy*, ktoré sú na konci spojené v jednom samostatnom vlákne a obalené príslušnou hlavičkou a trailerom.

Východzia veľkosť komprimovaného bloku je 128KB, ale je možné ju zmeniť. Prednastavený počet vlákien, ktoré vykonávajú kompresiu je 8 ale je možné ich zmeniť. Vstupné bloky sú komprimované závisle na sebe ale majú k dispozícii 32KB slovníku z predchádzajúceho bloku aby sa zachovala účinnosť kompresie. Táto funkcionalita môže byť vypnutá takže bloky sa budú komprimovať nezávisle na sebe.

8.3 Paralelizácia

Pigz používa na synchronizáciu vlákien knižnicu POSIX thread a pomocou rozhrania *yarn.h* . môže byť nahradená ekvivalentnou implementáciou *yarn.c*, ktorá používa inú knižnicu pre vlákna.

Pri paralelnej kompresii algoritmus použije hlavné vlákno na čítanie vstupných blokov a vloží ich do kompresného *job listu.* Pre každý blok je určené sekvenčné číslo, ktoré určuje jeho správne poradie. Každé kompresné vlákno pokračuje vo vykonávaní úloh (*jobs)*, pokým nedostane signál k ukončeniu.

Keď sa začína spracovávať úloha z *job listu,* tak sa do DEFLATE algoritmu načíta slovník(v prípade, že je dostupný). Následne sú vstupné dáta skomprimované na výstupný buffer. Úloha je potom uložená do zapisovacieho *job listu*, kde sú zoradené všetky úlohy podľa poradia. Kompresné vlákno aj naďalej pokračuje vo výpočte kontrolnej hodnoty pre výstupne dáta, a to buď *CRC-32* alebo *Adler-32 a* pokiaľ je to možné tak paralelne so zapisovacím vláknom. Akonáhle je toto hotové, kompresné vlákno uvoľní buffer a zámok na kontrolnej hodnote, takže zapisovacie vlákno ju môže začať kombinovať s predchádzajúcimi kontrolnými hodnotami. Kompresné vlákno teraz dokončilo svoju úlohu, takže môže vykonávať ďalšiu. Všetky kompresné vlákna sú ponechané v behu a čakajú aj potom, čo bol skomprimovaný posledný blok vstupného súboru a to pre prípad, žeby na vstupe(príkazový riadok) boli ešte ďalšie súbory.

Pred tým než sa začne čítať vstup, hlavné vlákno spustí zapisovacie vlákno aby bolo hneď pripravené spracovať úlohu. Kompresné vlákno vkladá zapisovacie úlohy do usporiadaného zoznamu, takže prvá úloha ma najnižšie poradové číslo. Zapisovacie vlákno čaká na ďalšie a ďalšie úlohy v zozname. Každá z týchto úloh si ešte drží svoj buffer pre kombináciu kontrolnej hodnoty. Potom zapisovacie vlákno vymaže pamäť a je pripravené na ďalšie použitie.

Držanie vstupnej vyrovnávacej pamäte pokiaľ nenaštartuje zapisovacie vlákno má výhodu v tom, že čítacie a kompresné vlákno sa neodstane ďaleko od zapisovacieho vlákna a nevznikne veľký počet nezapísaných skomprimovaných dát. Zapisovacie vlákno bude zapisovať skomprimované dáta, vymaže vstupnú pamäť a potom čaká na odomknutie kontrolnej hodnoty kompresným vláknom. Potom zapisovacie vlákno kombinuje kontrolnú hodnotu pre aktuálne komprimovaný blok s celkovou kontrolnou hodnotou pre prípadne použite v traileri(je to časť komprimovaného bloku a takisto ako *CRC-32* slúži na identifikáciu chýb). Ak to nie je posledný blok tak zapisovacie vlákno ide vykonávať ďalší blok v sekvencii. Po zápise posledného bloku sa toto vlákno vráti do hlavného vlákna. Na rozdiel od kompresného vlákna je zapisovacie vlákno spustené pre každý nový vstup. Zapisovacie vlákno zapisuje príslušnú hlavičku a blok dát ku skomprimovaným dátam.

Vstupné a výstupné buffery sa používajú opakovane skrz úlohy, ktoré sú v zoznamoch(*pools*). Každý buffer má počet použiteľnosti, ktorý ak je dekrementovaný na 0 tak ho vráti

do príslušného *job listu.* Každý vstupný buffer ma až tri paralelné využitia: vstup pre kompresiu, dáta pre výpočet kontrolnej hodnoty a ako slovník pre kompresiu. Každý výstupný buffer má len jedno použitie a to dáta, ktoré budú sériovo zapísané. Veľkosť vstupného *job listu* je obmedzená počtom bufferov, takže sa čítanie dát nedostane ďaleko pred kompresiu a nebude spotrebovávať pamäť. Limit sa rovná približne dvojnásobku počtu kompresných vlákien. V prípade, že čítanie je v porovnaní s kompresiou rýchlejšie tak tento počet umožňuje druhú sadu bufferov zatiaľ, čo bude prvá komprimovaná. Počet vstupných pamätí nie je priamo obmedzený, ale je nepriamo obmedzený počtom uvoľnených vstupných bufferov teda približne ten istý počet.

8.3.1 **Dekompresia**

Dekompresia nemôže byť paralelizovaná ľubovoľným počtom procesorov tak ako kompresia. Pre tento účel by musel byť špeciálne upravený *deflate stream.* V dôsledku toho pigz používa iba jedno vlákno, ale pre zápis, čítanie z disku a výpočet kontrolnej hodnoty sa vytvárajú samostatné vlákna.

9 Paralelizácia bzip2

Bzip2 je na paralelizáciu veľmi vhodný algoritmus. A to z dôvodu, že už vo svojej sekvenčnej verzii rozdeľuje vstupné dáta na rovnaké bloky(kapitola Bzip2). Pre bližšie zoznámenie s technológiou openMP bola zvolená paralelizácia bzip2.

9.1 Návrh a implementácia

Pre kompatibilitu s bzip2 a jednoduchosť bola použitá voľne dostupná knižnica *libbzip2*, ktorá umožňuje kompresiu konkrétneho bloku dát(kapitola Konzument).

V prvom kroku sa načíta celý súbor do pamäte.(V dnešnej dobe disponujú počítače dostatkom operačnej pamäte ale v prípade veľkosti súboru v radoch desiatok GB by tu nastal problém). Následne sa určí počet blokov, na ktoré sa rozdelí súbor – veľkosť súboru vydelíme veľkosťou jedného bloku. Veľkosť bloku je možné prispôsobiť. V ďalšom kroku inicializujeme vektor *std::vector<outputBuff>,* ktorého veľkosť bude počet blokov a každý blok bude obsahovať skomprimované dáta a ich veľkosť. Teraz príde na rad paralelizácia pomocou direktívy(viac kap. openMP):

```
#pragma omp parallel for shared(FileData,OutputBuffer,i) num_threads(t)
```

`shared` - tieto premenné budú zdieľané medzi všetkými vláknami

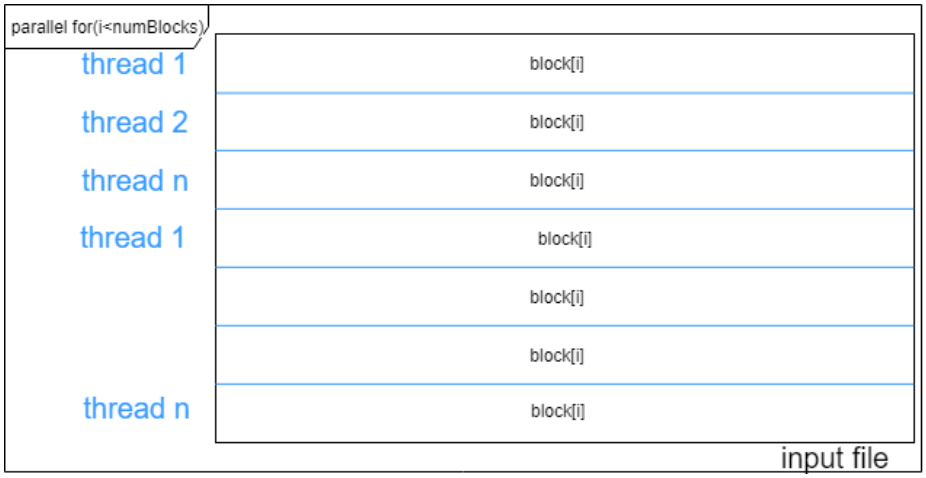
`FileData` – celý vstupný súbor v pamäti

`OutpuBuffer` – pole(vektor), do ktorého sa ukladajú skomprimované dáta

`i` – index iterácie v cykle for

`t` – počet vlákien

Počet iterácii v tomto cykle je rovný počtu blokov. Hneď na začiatku je podmienka, ktorá zaručuje, že k poslednému komprimovanému bloku sa pridá zvyšok súboru, ktorý vznikol pri celočíselnom delení. Na základe zdieľaného indexu sa postupne nastavuje blok, ktorý sa skomprimuje a pomocou tohto indexu sa daný blok uloží do výsledného vektoru, čím sa zabezpečí, že skomprimované bloky budú v správnom poradí obr. 6.



Obrázok 6: *Čítanie súboru pomocou openMP*

Nakoniec sa v jednom cykle prejde naplnený vektor a bloky sa postupne zapíšu na disk.

9.2 Optimalizácie a vylepšenia

Keďže sa jednalo o demonštráciu technológie a algoritmu samotná implementácia by sa dala optimalizovať a zrýchliť. Kvôli prehľadnosti a čitateľnosti sú vytvárané pomocné premenné, ktoré zbytočne zvyšujú nároky na pamäť. Vstupný súbor nemusí byť v pamäti celý ale môže sa čítať postupne a výsledok priamo predať kompresnej funkcie. Zápis do súboru by mohol prebiehať simultánne s kompresiou. Implementácia je súčasťou práce.

9.3 Ďalšie možnosti paralelizácie bzip2

Ďalším prístupom ako možno zrýchliť tento algoritmus je zamerať sa na paralelizáciu jeho jednotlivých krokov tzn. Burrows-Wheelerovu transformáciu a Huffmanovo kódovanie.

9.3.1 Paralelizácia BWT

V prvom kroku transformácie sa vytvoria všetky cyklické rotácie vstupného reťazca. Toto je možné implementovať pomocou jedného cyklu. Keďže všetky rotácie sú v ďalšom kroku zoradené podľa abecedy nezáleží na tom v akom poradí budú pridané do výsledného vektora. Tým pádom je tu vhodné použite openMP. Problémom je vkladanie rotácií do zdieľaného vektoru. Prvou možnosťou je použiť `#pragma omp critical`, čo zabezpečí, že k vektoru bude mať prístup iba jedno vlákno. Efektívnejšie(rýchlejšie) je použite `concurrent_vector`, ktorý je *thread-safe*. Pri malom vstupe je program veľmi neefektívny. Zdrojový kód by potom vypadal nasledovne:

```
string BWT_encode(string source) {
	//pridanie špecalného zanku, ktorý označuje koniec reťazca
	source.append('$');
	//vektor do, ktorého sa ukladajú rotácie
	Concurrency::concurrent_vector<string> shifts;
// iterácia cez vstupný reťazec
# pragma omp parallel for shared(source,shifts) num_threads(t)
	for( int i = 0; i < source.length(); i++) {
		string shift = cyclic_shift(source, i);
		shifts.push_back(shift);
	}

	//zoradenie podľa abecedy
	sort(shifts.begin(), shifts.end());

//na výstupe bude reťazec zložený z posledného znaku každej rotácie
	string encoding = "";

for (Concurrency::concurrent_vector<string>::iterator it =
shifts.begin(); it != shifts.end(); it++) {
		encoding = encoding + it->at(it->length() - 1);
	}
	return encoding;
}

//vytvotrenie rotácie
string cyclic_shift(string source, int distance) {
	return source.substr(distance, string::npos) + source.substr(0,
distance);
}
```

9.3.2 **Paralelizácia Huffmanovho kódovania**

Znaky, vyskytujúce sa vo vstupnom súbore najčastejšie, sa konvertujú na bitové reťazce s najkratšou dĺžkou. Toto kódovanie má vlastnosť prefixu - je jednoznačne dekódovateľné. Kompresia sa skladá z niekoľkých krokov:

1. Prejdeme celý súbor a zistíme početnosť(pravdepodobnosť) jednotlivých znakov, ktoré sú v súbore
2. Zo získaného histogramu vytvoríme Huffmanov(binárny) strom
3. Zo stromu získame prefixovú kódovú tabuľku
4. Podľa tejto tabuľky zakódujeme text

Hneď prvý krok je možne veľmi efektívne rozdeliť medzí viac vlákien. Tak ako v kapitole 10.1 sa vstup rozdelí na bloky a v každom bloku pomocou jedného vlákna sa spočítajú početnosti, keď dobehnú všetky vlákna všetky tieto početnosti sa zlúčia do výsledného histogramu. Týmto sa zrýchli beh a nebude to mať vplyv na výsledný kompresný pomer a je možné získať lineárne zrýchlenie. Táto paralelizácia bude mať zmysel iba pri dostatočne veľkých súboroch, pretože tvorenie výsledného histrogramu a réžia openMP synchronizácie zaberie určitý čas.

Rovnaký prístup je možné zvoliť aj pri štvrtom kroku. Rozdelenie vstupu do blokov, kde každý blok bude kódovať jedno vlákno. Do výsledného súboru je potrebné zapísať informácie o veľkosti jednotlivých blokov. Beh sa zrýchli ale môže dôjsť k zhoršeniu kompresného pomeru.

10 Paralelizácia XZ

XZ Utils je bezplatný univerzálny softvér na kompresiu dát s vysokým pomerom kompresie. XZ boli napísané pre POSIX-like systémy, ale pracujú aj na niektorých ne-POSIX systémoch. XZ sú nástupcom LZMA Utils. XZ vytvára o 30% menšie výstupy než gzip a o 15% menší výkon ako bzip2 Autorom je Lasse Collin [19].

10.1 XZ a LZMA SDK

Jadro kódu XZ je založené na LZMA SDK(7-zip), ale bolo do značnej miery upravené, aby bolo vhodný pre XZ. Primárny algoritmus kompresie je v súčasnosti LZMA2, ktorý sa používa vo formáte kontajnera .xz. Obidva nástroje podporujú rovnaký formát súborov, a preto sú interoperabilné. Autor 7zipu Igor Pavlov tvrdí, že XZ je dobré na dve veci - keďže je to alternatíva k 7-zipu bude jednoduchšie nájsť chybu v jednej z týchto verzií a je orientovaná viac na linuxové systémy[16]. Aj preto je súčasťou tejto práce.

10.1.1 7zip

Je ďalší komprimačný nástroj, ktorý podporuje viacvláknovú kompresiu. Ako jeden z mála je implementovaný pod OS Windows ale existuje k nemu linuxová alternatíva p7zip, ktorá ale nezodpovedá poslednej verzii 7zip. Jeho základom je algoritmus LZMA2, ktorý je nadstavbou LZMA. Jeho zdrojový kód je verejný ale dosť roztrieštený. Obsahuje všetku prácu, ktorú na ňom Igor Pavlov vykonal. Práca s pamäťou je vysoko optimalizovaná a dokonca niektoré časti kódu sú napísane v assembleri. Vďaka týmto aspektom je kód veľmi ťažko analyzovateľný.

Jeho primárna paralelizácia spočíva v tom, že na kompresiu používa tri vlákna. Jedno vlákno robí *lz-match-hash*, druhé vlákno sa stará o *lz-match-find* a tretie robí samotné kódovanie. Pri kompresii pomocou štyroch a viac vlákien sa už vstup rozdelí na bloky a kompresia je rýchlejšia za cenu horšieho kompresného pomeru.

10.2 Návrh a implementácia

Aby bolo možné získať z klastra maximum bola pre paralelizáciu zvolená kombinácia MPI a openMPI. Obidve technológie boli už spomenuté. V čase písania tejto práce zatiaľ nebol tento hybridný prístup nikde použitý. Implementácia bola inšpirovaná návrhom Lasse Collina[19].

V prvom rade bolo treba zistiť či neexituje nejaký spôsob, ktorý umožní čítať zo súboru paralelne. To znamená, že sa súbor nebude čítať postupne alebo že sa celý načíta do pamäte. Ako najvhodnejší prístup bol zvolený MPI IO.

Na začiatku bolo potrebné definovať dátové typy. MPI definuje vlastné dátové typy, ktoré sú potrebné pre I/O(vstupno/výstupné) operácie. V tomto prípade boli použité :

MPI_File - je handler súboru – MPI alternatíva pre dátový typ FILE

MPI_Offset – celočíselný dátový typ dostatočne veľký nato aby reprezentoval veľkosť(v bajtoch) najväčšieho súboru podporovaného MPI.

V ďalšom kroku sa už vykonávaný kód rozdelil medzi jednotlivé procesy. Bolo nutné zabezpečiť aby sa názov súboru dostal do všetkých procesov. Tento proces je zložený z dvoch krokov a využíva nato funkciu:

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
                  int root, MPI_Comm comm)
```

`buffer` – ukazateľ na adresu kde je uložená hodnota

`count` – veľkosť uloženej hodnoty

`datatype` – dátový typ hodnoty

`root` – číslo hlavného procesu (master)

`comm` – komunikátor (kde sa ma správa poslať)

Táto funkcia patrí do kolektívnej komunikácie. Funguje tak, že master pošle správu všetkým ostatným procesom. Na rozdiel do `MPI_Sned` nie je potrebné túto správu prijať. Funkcia funguje aj ako príjemca.

Keďže je potrebné poslať názov súboru všetkým procesom je nutné poznať aj jeho dĺžku. Takže najprv pošleme všetkým procesom dĺžku názvu vstupného súboru.

```
MPI_Bcast(&inputFileNameLength, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
```

Následne alokujeme veľkosť a pošleme názov súboru:

```
inputFileName = (char *) malloc(inputFileNameLength);
```

```
MPI_Bcast(inputFileName, inputFileNameLength, MPI_CHAR, MASTER_RANK,
MPI_COMM_WORLD);
```

Synchronizujeme všetky procesy pomocou `MPI_Barrier(MPI_COMM_WORLD)`. Táto funkcia zabezpečí, že všetky procesy budú čakať kým nedobehne posledný z nich.

Teraz každý z procesov pristúpi k otvoreniu súboru pomocou funkcie :

```
int MPI_File_open(MPI_Comm comm, const char *filename,
    int amode, MPI_Info info, MPI_File *fh)
```

`comm` – komunikátor

`filename` – názov súboru

`amode` – spôsob prístupu (read, write…)

`info` – informácie o objekte

`fh` – výstupný parameter, ktorý nastaví suborový handler

Pomocou funkcie `MPI_File_get_size(inputFh, &fileSize)` zistíme veľkosť súboru. Na základe tejto veľkosti rozdelíme súbor podľa počtu procesov na nezávislé bloky. Posledný blok bude mať takú veľkosť aby bol rozdelený celý súbor. Keďže program už beží v MPI nastaví sa offset súboru podľa poradia procesu, v ktorom sa momentálne nachádza . Veľmi dôležitým krokom je aktualizovať ukazatele na súbor pre jednotlivé procesy. Na to slúži funkcia `MPI_File_seek(inputFh, fileOffset, MPI_SEEK_SET)`, ktorá nastaví procesu ukazateľ na konkrétne miesto v súbore. V ďalšom kroku dochádza k samotnému čítaniu konkrétnej časti. Na to slúži funkcia:

```
MPI_File_read(MPI_File fh, void *buf,     int count,
    MPI_Datatype datatype, MPI_Status *status)
```

`fh` – handler súboru

`buff` – buffer, kde sa načítajú dáta

`cout` – veľkosť bufferu

`datatype` - dátový typ každého prvku v bufferi

`status` – status objektu

V tomto momente začínajú jednotlivé procesy nezávisle na sebe čítať svoju časť súboru, ktorá im bola nastavená.

V nasledujúcom kroku je volaná funkcia :

```
compress_lzma(char *mpiDataBlock, int mpiDataBlockSize,int
numberOfThreads, uint8_t** outputBlock, size_t& outputBlockSize )
```

`mpiDataBlock` – blok, ktorý sa bude komprimovať

`mpiDataBlockSize` – veľkosť komprimovaného bloku

`numberOfThreads` – počet vlákien pre openMP

`outputBlock` – skomprimované dáta (výstupný parameter)

`outpuBlockSize` – veľkosť výstupných dát

V tejto funkcii prebieha už samotná kompresia. Najprv sa nastaví parameter `lzma_options_lzma`. Týmto parametrom sa nastavujú možnosti kompresnej metódy. LZMA a LZMA2 zdieľajú väčšinu zdrojového kódu preto sa používa toto nastavenie pre obe tieto metódy s malými rozdielmi. Funkcia `lzma_lzma_preset` tieto nastavenie predá knižnici *liblzma* spolu s rýchlosťou kompresie. V tomto prípade je 0 najrýchlejšia 9 najpomalšia(najefektívnejšia). Nasleduje nastavenie veľkosti bloku, ktorý bude komprimovaný:

```
srcChunkSize = lzma.dict_size * 3
```

Po experimentovaní bolo zistené, že trojnásobok veľkosti bloku je najlepší kompromis medzi kompresným pomerom, spotrebou pamäte a výkonom. Pre veľkosť výstupného bufferu bol zvolený veľmi pesimistický vzorec, ktorý je ale pre demonštráciu postačujúci.

```
destChunkSize = srcChunkSize + srcChunkSize / 64 + 12
```

Vypočíta sa počet blokov na ktorý sa rozdelí vstup a prebehne kontrola, kde sa predíde situácii, žeby bolo použitých viac vlákien ako blokov.

Na rad prichádza najdôležitejšia časť paralelizácie cyklus *for,* kde prebieha samotná kompresia a je zapísaný pomocou konštruktu:

```
#   pragma omp parallel for ordered firstprivate(lzma) \
      num_threads(numberOfThreads) schedule(static, 1)
```

Direktíva *ordered* spôsobí, že jednotlivé iterácie budú za sebou nasledovať tak ako v klasickom cykle. Bez tejto direktívy nemôžeme zaručiť, že sa vlákna budú spúšťať postupne a tým pádom by nebolo možné čítať vstupný blok postupne.

Direktíva *firstprivate(lzma)* znamená že premenná *lzma* môže byť inicializovaná už pred použitím openMP a v každom vlákno bude mať rovnakú hodnotu.

Direktíva *schedule(static, 1)* má za úlohu rozdeliť jednotlivé vlákna(iterácie) do blokov. Toto nastavenie spôsobí, že pri 5 vláknach pôjdu iterácie v poradí 0,1,2,3,4 | 0,1,2,3,4… . Pri nastavení *schedule(static, 5)* by to bolo 0,0,0,0,0 | 1,1,1,1,1 .... 4,4,4,4,4.

Všetky tieto nastavenia sú potrebné pre *preset_dictionary.* V implementáci sa tento „*vopred nastavený slovník*“ nachádza hneď za direktívou *parallel for*:

```
lzma.preset_dict = &src[(i - 1) * srcChunkSize];
```

Nastavujeme z čoho sa bude skladať obsah slovníka a:

```
lzma.preset_dict_size = (uint32_t) srcChunkSize;
```

nastavujeme veľkosť slovníka.

Tento slovník funguje tak, že je v ňom uložená história okna algoritmu LZ77. To znamená, že aj pri rozdelení vstupu na bloky sa kompresný pomer nemusí vôbec zmeniť. Preto je veľmi vhodný na paralelizáciu. Z tohto dôvodu bol obsah a veľkosť slovníka nastavený na celý blok. V súčasnosti je ho možné použiť iba pri *raw* kompresii a dekompresii. V budúcností je plánovaná podpora aj pre klasické *.xz* a *.lzma* formáty. Jeho použitie je možné si predstaviť takto:

[Chunk 1][Chunk 2][Chunk 3]

Thread 1 ccccccccc

Thread 2 pppccccccccc

Thread 3 pppccccccccc

c - aktuálna kompresia

p - priradenie vyhľadávača zhody pomocou prednastaveného slovníka

Ďalej sa natavuje *lzma_filer* :

```
lzma_filter filters[2] = {
        { LZMA_FILTER_LZMA2, &lzma },
        { LZMA_VLI_UNKNOWN, NULL }
};
```

Pomocou tohto filtra sa nastavuje *reťaz(chain)* filtrov(kompresných metód), ktoré budú použité pri kompresii. `LZMA_FILTER_LZMA2` určuje, že bude použitá iba metóda LZMA2. `LZMA_VLI_UNKNOWN` je podľa špecifikácie označenie konca tejto reťaze filtrov. Nasleduje nastavenie vstupnej veľkosti bloku a inicializácia premenných pre výsledok, ktorý nastaví komprimačná funkcia :

```
lzma_raw_buffer_encode(
            const lzma_filter *filters, lzma_allocator *allocator,
            const uint8_t *in, size_t in_size, uint8_t *out,
            size_t *out_pos, size_t out_size)
```

`filters` – pole filtrov, ktoré sa budú aplikovať na dáta

`allocator` – špeciálne funkcie, ktoré sa môžu zavolať pri kompresii

`in` – začiatok vstupného bufferu

`in_pos` – ďalší byte bude čitaný z [*in_pos]

`in_size` – veľkosť vstupného bufferu

`out` – začiatok výstupného bufferu

`out_pos` – ďalší byte bude zapísaný na [*out_pos]

`out_size` – veľkosť výstupného súboru

Po kompresii sa pomocou direktívy `#pragma omp ordered`, ktorá zaručí, že sa jednotlivé vlákna budú volať postupne tzn. 0,1,2,3.., sa jednotlivé bloky spoja do jedného. Pomocou *MPI_Barrier* počkáme kým dobehnú všetky procesy. Teraz bude potrebné zapísať skomprimované dáta do výsledného súboru. Keďže MPI nemá k dispozícii žiadny príkaz, ktorý by procesy zoradil podľa ich *ranku* bolo nutné vymyslieť spôsob ako to dosiahnuť:

```
for(i = 0; i < poolSize; i++) {
            MPI_Barrier(MPI_COMM_WORLD);
            if (i == currentRank) {
                fwrite(outputBlock,sizeof(char),outputBlockSize,f);
                fclose(f);
            }
```

Cyklus prechádza postupne všetky procesy vždy ich zosynchronizuje a k zápisu pustí len ten, ktorý je v správnom poradí. Takáto implementácia zápisu je nie veľmi neefektívna. MPI má k dispozícii aj paralelné zapisovanie pomocou volania *MPI_Write_file*. V tomto prípade, kedy záleží na poradí blokov nie je možné využiť jej plný potenciál.

10.3 Použitie

Spustenie nástroja PCompress prebieha pomocou príkazu :

```
mpirun -np 2 PCompress - t 4 -f input.txt
```

MPI prostrediu týmto hovoríme aby spustil PCompress v dvoch procesoch a pre každý proces nastavil 4 vlákna pre openMP. Program vytvorí skomprimovaný súbor v tej istej lokácii s príponou **.xz*. Dekompresia prebieha veľmi rýchlo, čiže jej paralelizácia nebude potrebná a urobí sa pomocou príkazu:

```
xz -dc -F raw --lzma2=dict=8MiB INFILE > OUTFILE
```

XZ dekomprimuje súbor pomocou raw dekodéra s veľkosťou slovníka 8MB.

Pri kompresii je veľmi dôležité pozorovať, že každý MPI proces spustil správny počet MPI vlákien. Pri testovaní bolo zistené, že niektoré linuxové distribúcie(v distribúcii Ubuntu nebol nájdený spôsob ako to ovplyvniť) majú obmedzený počet vlákien pre jeden MPI proces. V praxi to znamená, že pokiaľ sa použije pre spustenie príkaz uvedený vyššie tak v resources monitore musí byť vyťažených 8 procesorov. V prípade, že budú vyťažené len 4 kompresia síce prebehne ale skomprimovaný súbor nebude možno úplne dekomprimovať, pretože bloky nebudú v správnom poradí. Je to spôsobené tým, že blok dát, ktorý sa skomprimuje, sa ukladá na adresu, ktorá sa počíta aj na základe počtu vlákien. Ak pri sa paralelizácii tento počet zmení a nastane problém.

11 Testy a analýza

Celá práca bola analyzovaná na operačnom systéme Linux Mint 18 "Sarah" x64. Hardware počítača sa skladal z procesora Intel Core i7-4702MQ s frekvenciou 2.20GHz. Operačná pamäť mala 16GB a pevný disk bol typu SSD.

11.1 Analýza

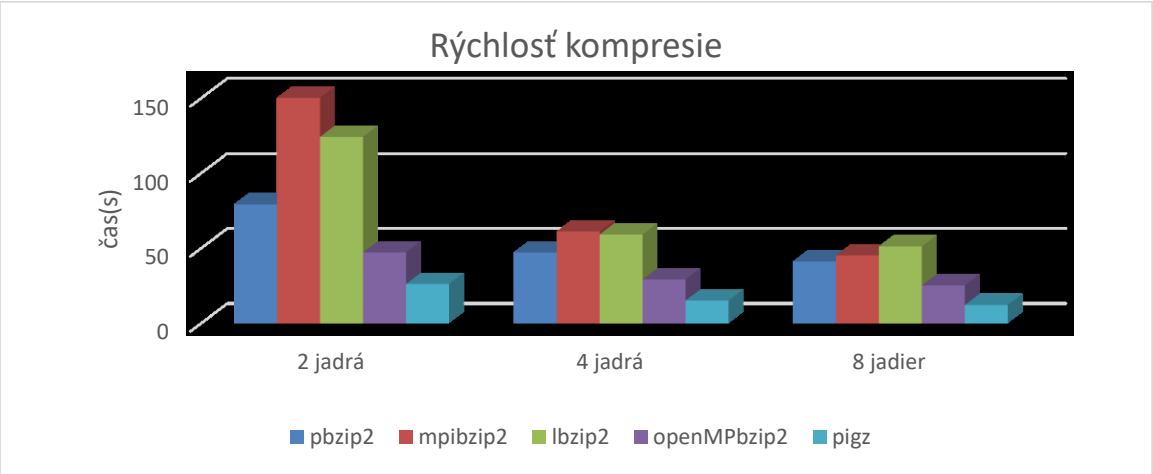
Skúmanie algoritmov prebehlo vo vývojom prostredí CLion od JetBrains, ktoré je najbližšia alternatíva k Visual Studio pod OS Linux pre vývoj v jazyku C/C++. Ponúka všetky nástroje pre prácu s vláknami a má veľmi dobru podporu *code completition*(inellisense).

Najnáročnejšie bolo jednotlivé zdrojové kódy spustiť aby ich bolo možné analyzovať krok po kroku. Bolo potrebné nainštalovať chýbajúce knižnice, upravovať hlavičkové súbory aby sedeli cesty a vždy správne zostaviť CMakeLists. Vo väčšine prípadov nebol kód takmer vôbec okomentovaný a nebola dostupná žiadna obsiahlejšia dokumentácia. Najviac ochoty spolupracovať preukázal doktor Jeff Gilchirst. Naopak najmenej ochotný bol Igor Pavlov, ktorého odpovede boli veľmi stručné a obecné.

11.2 Testy

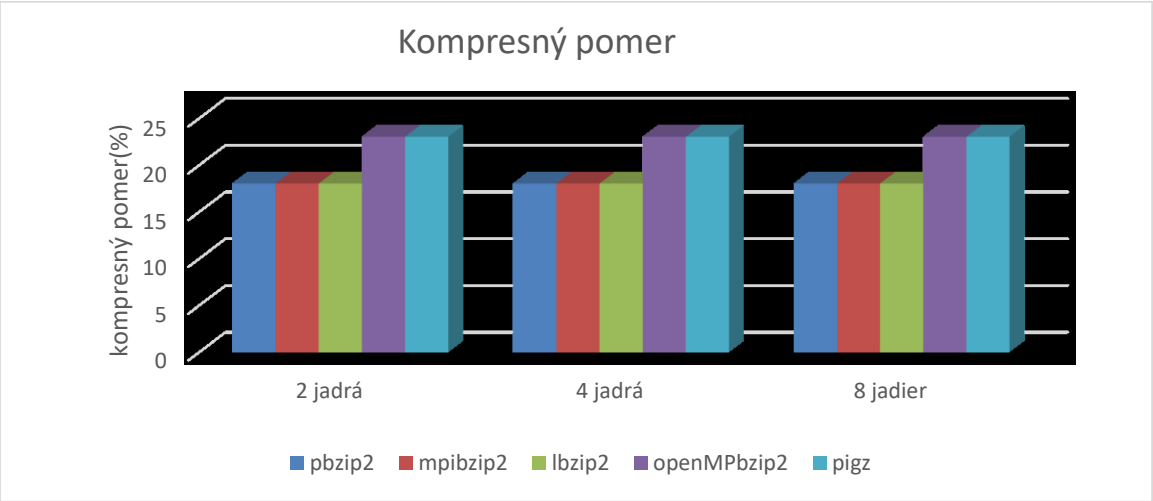
Niektoré algoritmy je možno priamo nainštalovať a používať cez terminál. Ostatné boli testované vo vývojom prostredí. Súbor bol 1GB z Wikipédie vo forme xml. Meral sa celkový beh programu tzn. , že aj čas zápisu na disk.

11.2.1 **Algoritmy bzip2 a deflate**



Obrázok 7: *Rýchlosť kompresie v závislosti od počtu jadier pre bzip2 a deflate*

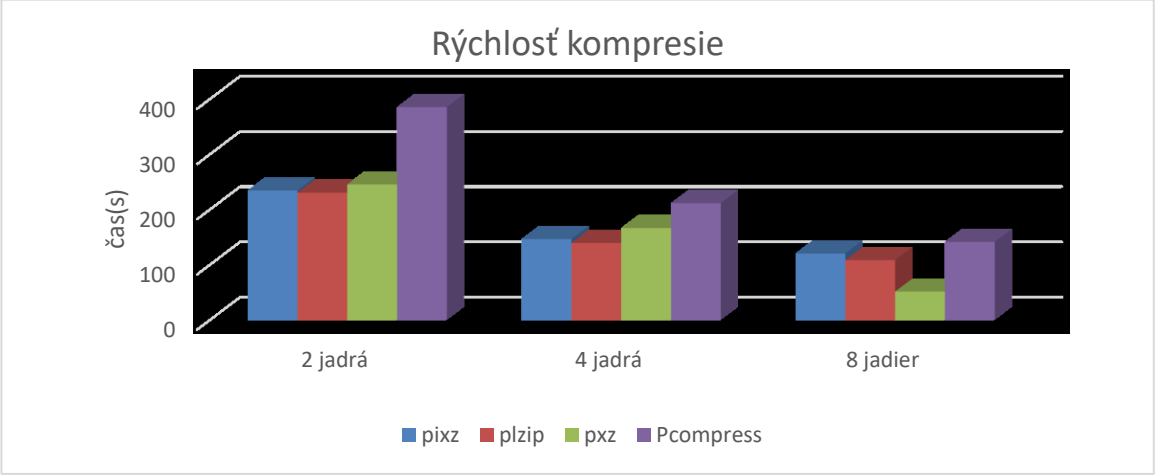
Z grafu možno vyčítať, že s počtom jadier sa rýchlosť algoritmu zvyšuje. V niektorých prípadoch dokonca lineárne. Ako najrýchlejší dopadol pigz. OpenMPbizp2 implementovaný v tejto práci skončil druhý.



Obrázok 8: *Kompresný pomer v závislosti počtu jadier pre bzip2 a deflate*

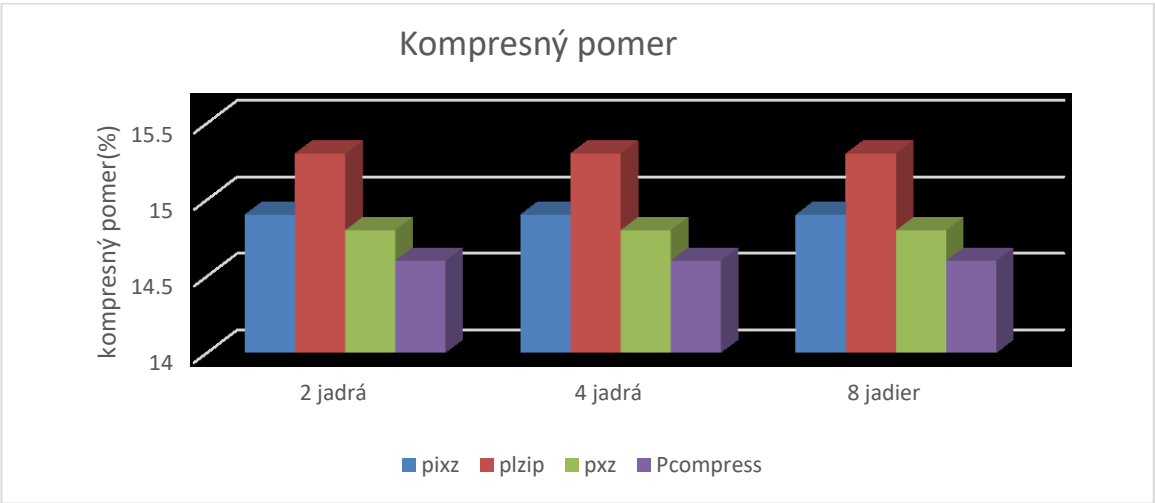
Kompresný pomer je vo všetkých algoritmoch rovnaký nezávisle od počtu jadier. Je to spôsobené tým, že v každom algoritme bola vždy nastavená rovnaká veľkosť bloku, ktorá sa bude komprimovať. Takže kompresný pomer sa zachová a zvýši sa len rýchlosť pretože, sa pracuje s viac blokmi zároveň. OpenMPbzip2 má nastavený menší blok tzn. je rýchlejší ale za cenu horšieho kompresného pomeru.

11.2.2 **Algoritmus lzma2**



Obrázok 9: *Rýchlosť kompresie v závislosti od počtu jadier pre lzma2*

Takisto aj tu je vidieť, že počet jadier zvyšuje efektivitu algoritmu v prípade pxz dokonca lineárne. PCompress implementovaný v tejto práci dopadol najhoršie. Tak isto ako mpibzip2. Spôsobené to môže byť tým, že testy prebiehali na klasickom procesore. Oproti algoritmom bzip2 a deflate je nárast času dosť znateľný.



Obrázok 10: *Kompresný pomer v závislosti počtu jadier pre lzma2*

Ani tu sa kompresný pomer v závislosti na počtu jadier nemení. Opäť to je spôsobené tým, že je vopred daná veľkosť komprimovaných blokov mení sa len počet jadier, ktoré tento blok komprimujú. V prípade PCompress sa veľkosť bloku mení ale algoritmus využíva *preset dicitionary* čo zaručuje zachovanie kompresného pomeru.

12 Záver

V práci sme čitateľa uviedli do problematiky paralelizácie v bezstratovej kompresii. Následne sme sa venovali detailnej analýze jednotlivých nástrojov, ktoré sú v dnešnej dobe k dispozícii. Jednalo sa o algoritmy typu bzip2, lzma2 a deflate. Počas analýzy sme zistili, že všetky prístupy používajú rozdelenie súboru na bloky. Tým pádom dochádza k zhoršeniu kompresného poemru. Som toho názoru, že to dôvodom je pôvodný návrh algoritmov či už bzip2 alebo LZ.

V tejto práci nie sú spomenuté všetky existujúce algoritmy ale len tie najpoužívanejšie resp najznámejšie. Existuje ešte mnoho ďalších náastrojov, niektoré sú len prepísané do iných programovacích jazykov. Sú pokusy využiť na kompresiu GPU.

Iba dva existujúce algoritmy používajú technológie openMP alebo MPI. Podľa môjho názoru je to spôsobené tým, že openMP a MPI sú dosť náročné technológie a prispôsobiť im tak komplexný problém ako je bezstratová kompresia nie je jednoduchá záležitosť. Preto boli v tejto práci implementované práve tieto technológie. Prvý pokus ukazuje ako by bolo možné použiť openMP na BWT. Ďalej je v práci teoreticky načrtnutá možnosť paralelizácie Huffmanovho kódovania. Práca obsahuje aj implementáciu bzip2 a openMP. Demonštráciou oboch technológií je hybridný prístup, kedy pomocou MPI a openMP paralelizujeme knižnicu liblzma, ktorá je súčasťou XZ utils.

Testy naznačujú, že sú dve možnosti. Buď bude zvolený rýchly algoritmus za cenu horšieho pomeru(bzip2,deflate) alebo lepší kompresný pomer za dlhší čas(lzma2). Implementovaný PCompress bol najhorší čo sa rýchlosti týka pri najmenšom počte jadier. Spôsobené to je pravdepodobne tým, že program nebol spúšťaný na klastri. Podobný výsledok má aj MPIbzip2.

Ako hlavný prínos tejto práce vidím v tom, že boli použité obe technológie, ktoré sa používajú v prostredí HPC. Ďalším prínosom je detailná analýza stavajúcich riešení, pretože voľné dostupné kódy majú veľmi slabú dokumentáciu. V budúcnosti by bolo dobré naviazať na paralelizáciu jednotlivých častí algoritmov. Ako to bolo napríklad v prípade BWT alebo Hufmannovho kódovania.

Počas práce sa mi veľmi rozšírili poznatky z oblasti bezstratovej kompresie, paralelnom programovaní a čo v praxi ocením asi najviac analýze neznámeho zdrojového kódu.

Na záver by som by som chcel dodať že v paralelizácii kompresie textových súborov sme podľa mňa, ešte nedosiahli maximum. Možno to chce zmeniť prístup ako sa na kompresiu textu pozeráme ale väčšia pravdepodobnosť je, že problém je v texte samotnom.

Literatúra

[1] J. Gilchrist - [Online] dostupné:
http://compression.ca/pbzip2/ [Cit. 20.08.2017]

[2] Bzip2, Wikipédia - [Online] dostupné
https://en.wikipedia.org/wiki/Bzip2 [Cit. 19.06.2018]

[3] J. Gilchrist - [Online] dostupné
http://compression.ca/mpibzip2/ [Cit. 10.11.2017]

[4] MPI, Wikipédia - [Online] dostupné
https://sk.wikipedia.org/wiki/Message_Passing_Interface [Cit. 19. 06. 2018]

[5] OpenMPI dokumentácia - [Online] dostupné:
https://www.open-mpi.org/doc/v3.1/ [Cit. 19.06.2018]

[6] OpenMPI dokumentácia kapitola 6 - [Online] dostupné:
https://www.open-mpi.org/faq/?category=debugging [Cit. 19.06.2018]

[7] LBZIP2 – [Online] dostupné
http://lbzip2.org/ [Cit. 19.06.2018]

[8] Jindřich Nový, PXZ - [Online] dostupné
https://jnovy.fedorapeople.org/pxz/ [Cit. 19.06.2018]

[9] open MP dokumentácia [Online] dostupné
https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf [Cit. 19.06.2018]

[10] Antonio Diaz Diaz , Plzip – [Online] dostupné
https://www.nongnu.org/lzip/plzip.html [Cit. 19.06.2018]

[11] PIXZ [Online] dostupné
https://github.com/vasi/pixz [Cit. 19.06.2018]

[12] Mark Adler, PIGZ – [Online] dostupné
https://zlib.net/pigz/ [Cit. 19.06.2018]

[13] Mark Adler, Wikipedia – [Online] dostupné
https://en.wikipedia.org/wiki/Mark_Adler [Cit. 19.06.2018]

[14] GZIP, Wikipedia – [Online] dostupné
https://en.wikipedia.org/wiki/Gzip [Cit. 19.06.2018]

[15] P. Deutsch, DEFLATE Compressed Data Format Specification kapitola 2 - [Online] dostupné https://tools.ietf.org/html/rfc1951 [Cit. 19.06.2018]

[16] Igor Pavlov, SourceForge forum XZ discussion – [Online] dostupné https://sourceforge.net/p/lzmautils/discussion/708858/thread/3b2bddfb/ [Cit. 16.06.2018]

[17] History of lossless data compression algorithms [Online] dostupné http://ethw.org/History_of_Lossless_Data_Compression_Algorithms [Cit. 28.06.2018]

[18] OpenMP dokumentácia kapitola 1.2.1 - [Online] dostupné: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf [Cit. 28.06.2018]

[19] XZ utils - [Online] dostupné https://tukaani.org/xz/ [Cit. 28.06.2018]

[20] ZStandart – Release v1.1.3 - [Online] dostupné: https://github.com/facebook/zstd/releases

A Prílohy

Na priloženom cd nájdete tieto prílohy zazipované vo formáte .zip:

- projektové zložky pre vývojové CLION
    1. DP_pbzip2
    2. mpbzip2
    3. lbzip2
    4. openMPBzip2
    5. PCompress
    6. pigz
    7. pixz-master
    8. plzip
    9. pxz
- solution pre Visual Studio
    1. paBWT